

C# 12

— for —

Cloud, Web, and Desktop Applications

Modern concepts and techniques for
software development with C# 12



Thiago Vivas de Araujo



C# 12 for Cloud, Web, and Desktop Applications

*Modern concepts and
techniques for
software development with
C# 12*

Thiago Vivas de Araujo



www.bpbonline.com

First Edition 2024

Copyright © BPB Publications, India

ISBN: 978-93-55519-023

All Rights Reserved. No part of this publication may be reproduced, distributed or transmitted in any form or by any means or stored in a database or retrieval system, without the prior written permission of the publisher with the exception to the program listings which may be entered, stored and executed in a computer system, but they can not be reproduced by the means of publication, photocopy, recording, or by any electronic and mechanical means.

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true to correct and the best of author's and publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.

**To View Complete
BPB Publications Catalogue
Scan the QR Code:**



www.bpbonline.com

Dedicated to

My beloved parents:

Carlos Roberto Coutinho de Araujo

Sonia Maria Pereira Vivas

About the Author

Thiago is a Brazilian expert passionate about software development and the complexity that comes with the process of constructing reliable software. Thiago has over 12 years of experience in software development and has worked in big companies like Oi, Accenture, Rio 2016 Olympic Games, Altice Portugal, Vodafone, and Microsoft.

Thiago, as a content creator, has already published more than 60 technical articles that have helped more than 2 million knowledge-seekers, writing mainly about C#, DevOps, and Cloud development.

About the Reviewers

- **Tomasz Kocurek** is a .NET developer. He graduated in Applied Computer Science in the Faculty of Electrical Engineering, Automatics, Computer Science, and Electronics at AGH University of Cracow. He has worked in various industries: artificial intelligence, creating bots, cryptocurrencies, virtual reality, medicine, electronic document circulation, data protection, transport, and navigation. He is keen on artificial intelligence, especially in healthcare. His master's thesis was blood detection using machine learning. He loves reading books. He has tried to find something interesting in every part of science. He likes to talk about new technologies. He has created documents, wiki, and a few applications with information about projects because data is priceless. He is interested in mathematics, playing bridge and football.

- **Denis Kazakov** has spent over 17 years in the software industry with a breadth of experience in different businesses and environments, from corporations to startups and freelance.

He started as a developer and has worked through technical and development leadership roles. Today Denis's focus is on cloud technologies and development using Azure and .NET. Denis is the founder of the educational startup Similearn.com, where he works now. Denis regularly contributed to the IT community via forums of developers, open source projects, and blog posts. Denis has been awarded by Microsoft as a Most Valuable Professional four times in the Azure category

- **Dirk Strauss** is a seasoned software developer with over 19 years of experience utilizing C# and Visual Studio. He had the privilege of working with several companies and learning from some of the most talented individuals in the industry. In addition to his professional experience, he has published multiple books on topics such as C#, Visual Studio, and

ASP.NET Core. His passion for programming is unwavering, and he remains dedicated to staying current with the latest technology while sharing his expertise with others.

Acknowledgement

I am immensely grateful to my beloved parents and sisters for their unconditional support. Their encouragement has sustained me through the ups and downs of this journey. Despite their bewilderment at the intricacies of software development, their unwavering belief in me has been a guiding light. Thanks for everything; I love you.

To my friends from Lisbon, Portugal, thank you for understanding my occasional disappearance while I was lost in the depths of this book. Your patience, camaraderie, and occasional provision of beer made the journey more bearable and the late nights more enjoyable. Cheers to each of you for being part of this adventure; the next drinks round is on me!

To my childhood friends from Complexo da Chumbada, Rio de Janeiro, Brazil, your unwavering support and camaraderie throughout the years have been a source of immense strength and inspiration. From the streets of our neighborhood to the pages of this book, your friendship has been a guiding light. Thank you for the countless memories, laughter, and shared adventures that have shaped who I am today. Your friendship is truly cherished, and I am grateful for the bond we share.

Preface

This book covers many different aspects of web scraping, the importance of automation of web scraping. This book also introduces the importance of web scraping in the field of real time industry. It shows how the data is important for the industries. This book solves the basic understanding of web scraping / crawling in the data world. It also gives importance for python learning. So that python's basic concepts get refreshed. This book gives information about the usefulness of Python in web scraping as well.

This book takes a practical approach for web scraping learners. It covers few real time industry examples as well. It will cover information such as Python Basically used for automation, machine learning. It can be used for easy data manipulating and transforming. Used in different domains for data mining purposes. You can design API access frameworks for end to end automation in different domains like finance, retail etc.

This book is divided into 14 chapters. They will cover Python basics, basics and advance in web scraping, why python is better for web scraping, real time examples in web scraping etc. So, learners can get more interest in web scraping tools as well. The details are listed below:

Chapter 1: Introduction to Visual Studio 2022 - Readers will explore the redesigned user interface, enhanced code analysis tools, and new debugging and testing functionalities. This chapter aims to equip developers with the knowledge and tools needed to leverage Visual Studio's latest features for enhanced productivity and efficiency.

Chapter 2: What is New in C# 12 - Whether experienced or new to C#, this chapter provides valuable insights into the evolving C# landscape, empowering developers to leverage the latest tools and advancements for efficient and effective coding.

Chapter 3: Mastering Entity Framework Core - It explores Database-First and Code-First approaches, covering schema creation and data model generation. It explains data modeling concepts, data annotations, and customization. The chapter also addresses data management tasks like querying and modifying data using LINQ to Entities, with examples.

Chapter 4: Getting Started with Azure Functions - covers triggers and bindings, exploring HTTP, Timer, Blob, and Queue triggers with detailed setup instructions and best practices. The chapter also explains how bindings facilitate interaction with Azure services like Blob Storage and Cosmos DB, demonstrating data access and manipulation through binding examples.

Chapter 5: Azure SQL, Cosmos DB and Blob Storage - providing step-by-step instructions and key concepts for each service. From creating and connecting to Azure SQL Database instances to understanding Cosmos DB APIs and working with Blob Storage containers, readers will gain valuable insights into leveraging these services effectively within their .NET applications.

Chapter 6: Unleashing the Power of Async Operations with Azure Service Bus - explores key concepts like queues, topics, and subscriptions, providing practical instructions and best practices for implementation. From understanding messaging patterns to configuring advanced features like message sessions and dead-letter queues, readers will gain valuable insights into building robust and scalable messaging solutions with Azure Service Bus.

Chapter 7: Securing Your Apps with Azure Key Vault - focus on authentication, the chapter guides readers through creating and configuring Key Vault instances, storing secrets, and accessing sensitive data securely. Step-by-step instructions and coverage of authentication options, including client certificates and Azure Active Directory authentication, equip developers with the knowledge and tools to ensure robust security measures for their applications.

Chapter 8: Building Dynamic Web Apps with Blazor and ASP.NET - Focusing on Blazor's key features like Hot Reload, Security, and Strongly-typed databinding, the chapter presents practical case studies to demonstrate their application in real-world scenarios. Starting with an overview of Blazor and its benefits, it

covers project setup with ASP.NET and Visual Studio, exploring both client-side and server-side hosting models

Chapter 9: Real-time Communication with SignalR and ASP.NET - Starting with an overview of SignalR's benefits, it covers project setup with ASP.NET and Visual Studio, exploring various transport protocols like WebSockets and Server-Sent Events

Chapter 10: Implementing MicroServices with Web APIs - covers microservices architecture and the advantages of using Web APIs. It details designing scalable architectures, including service discovery, load balancing, and fault tolerance. Exploring scaling techniques like horizontal, vertical, and auto-scaling, it provides practical instructions, best practices, and pitfalls to avoid, empowering developers to create robust microservices architectures.

Chapter 11: CI/CD with Docker and Azure DevOps - Beginning with an overview of Docker and Azure DevOps, it demonstrates how to automate the CI-CD process. Covering Docker commands for building and deploying containerized apps, it explains CI-CD concepts and their role in streamlining development and deployment. The chapter walks through configuring build pipelines, release pipelines, and environment management using Azure DevOps

Chapter 12: Building Multi-platform Apps with .NET MAUI and Blazor Hybrid - explores its role in creating apps for various platforms. The chapter introduces Blazor Hybrid and its integration with .NET MAUI, comparing it with Blazor to highlight their respective benefits and drawbacks. It explains how Blazor Hybrid enables developers to build responsive UIs seamlessly integrated with .NET MAUI

Chapter 13: Windows UI Library: Crafting Native Windows Experience - Starting with an introduction to WinUI, it delves into how it facilitates the development of responsive desktop apps. Exploring the Fluent Design System's principles, including depth and motion, it demonstrates using WinUI's UI controls to implement engaging user interfaces

Chapter 14: Unit Testing and Debugging - craft effective unit tests using NUnit and xUnit for code reliability. Discover automated testing frameworks for better code quality. Dive into debugging

with C# tools, mastering techniques like breakpoints and variable inspection for efficient error resolution, ultimately boosting developer productivity

Code Bundle and Coloured Images

Please follow the link to download the *Code Bundle* and the *Coloured Images* of the book:

<https://rebrand.ly/fcn7t93>

The code bundle for the book is also hosted on GitHub at <https://github.com/bpbpublications/CSharp12-For-Cloud-Web-and-Desktop-Applications>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at <https://github.com/bpbpublications>. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.bpbonline.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at :

business@bpbonline.com for more details.

At www.bpbonline.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at business@bpbonline.com with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit www.bpbonline.com. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit www.bpbonline.com.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



Table of Contents

1. Introduction to Visual Studio 2022

Introduction

Structure

Objectives

Significant changes from Visual Studio 2019

Performance improvements

64 -bit support

Smarter IntelliCode

Razor editor improvements

Hot reload with new capabilities

Multi-repository support

Interactive staging support

Interface improvements and customization

Live unit testing

Using the live unit testing

Supported testing frameworks

Exclude and include test projects and test methods

Configuring according to your needs

Snapshot debugging

Using snapshot debugging

Supported frameworks and environments

Required permissions and limitations

Time Travelling Debugging

Using time travelling debug

Recording a snapshot

Viewing the snapshots

Time travelling debug limitation

Conclusion

2. What is New in C# 12

Introduction

Structure

Objectives

C# 11 updates

Raw string literals

Generic math support

Generic attributes

Unicode Transformation Format-8 string literals

Newlines in string interpolation expressions

List patterns

File-local types

Required members

Auto-default structs

Pattern match `Span<char>` on a constant string

Extended nameof scope

Numeric `IntPtr` and `UIntPtr`

ref fields and scoped ref

Improved method group conversion to delegate

Warning wave 7

C# 12 updates

Primary constructors

Collection expressions

Inline arrays

Default lambda parameters

ref readonly parameters

Alias any type

Conclusion

3. Mastering Entity Framework Core

Introduction

Structure

Objectives

Mastering Entity Framework Core

Database First

Benefits of using Database First

Implementation step-by-step

Code First

Benefits of using Code First

Implementation step-by-step

LINQ to Entities

Data Annotations

Most common Data Annotations

Applying Data Annotations

Data Modeling

One-to-one relationship

Required one-to-one

Optional one-to-one

One-to-many relationship

Required one-to-many

Optional one-to-many

Many-to-many relationship

Data Management

Normal usage

Unit of work

Repository pattern

Conclusion

4. Getting Started with Azure Functions

Introduction

Structure

Objectives

Getting started with Azure Functions

Azure Function triggers

Azure Function bindings

Practical case-study

Creating the Azure Function

Selecting the trigger

Picking an appropriate binding

Testing the output

Conclusion

Points to remember

Exercises

5. Azure SQL, Cosmos DB and Blob Storage

Introduction

Structure

Objectives

Azure SQL, Cosmos DB and Blob Storage

Azure SQL

Scaling Azure SQL Server

Usage example

Creating the Azure resource

Connecting to the database

Cosmos DB

Cosmos DB Containers

Scaling Cosmos DB

Triggers, stored procedures, and UDFs

Change feed notifications with Cosmos DB

Usage examples of Cosmos DB for NoSQL

Creating the Azure Resource

Connecting to the Database

Blob Storage

Scaling Blob Storage

Usage example

Creating the Azure Resource

Connecting to the Database

Conclusion

6. Unleashing the Power of Async Operations with Azure Service Bus

Introduction

Structure

Objectives

Async operations with Service Bus

Azure Service Bus Queues

Session Queues

Azure Service Bus Topics

Azure Service Bus Subscriptions

Azure Service Bus vs Azure Queue Storage Queues

Case study

Creating a new Azure Service Bus

Publishing to Azure Service Bus

Consuming messages

Consuming message batches

Message processor

Consuming sessions

Session processor

Consuming Topics and Subscriptions

Conclusion

7. Securing Your Apps with Azure Key Vault

Introduction

Structure

Objectives

Azure Key Vault Overview

Azure Key Vault Authentication

Azure Key Vault Access policies

Case study

Creating Azure Key Vault

Managing Key Vault Access policies

Accessing a key

Conclusion

8. Building Dynamic Web Apps with Blazor and ASP.NET

Introduction

Structure

Objectives

Web Apps with Blazor and .NET

Hot reload

Supported frameworks and application types

Unsupported Scenarios

Configuring Hot Reload

Security

Blazor server authentication

Blazor WebAssembly authentication

Authorization

AuthorizeView component

Authorize attribute

Data binding

Two-way data binding

Chained data binding

Blazor vs Razor

Best practices

Practical case study

Creating a Blazor Server App project

Working of hot reload

Authorization and authentication

Data binding

Conclusion

9. Real-time Communication with SignalR and ASP.NET

Introduction

Structure

Objectives

Real-time communication with SignalR and ASP.NET

Where to use SignalR

Message transports

Hubs

SignalR methods

Configuration

JSON encoding

MessagePack encoding

Server configuration options

Advanced HTTP configuration

Client configuration options

Configure logging

Configure allowed transports

Configure Bearer authentication

Additional options

Authentication and authorization

Cookie authentication

Bearer token authentication

Identity server JWT authentication

Windows authentication

Claims

Policies for hubs and hubs methods authorization

Authorization handlers

Streaming hub

Server-to-client streaming hub

Client-to-server streaming hub

Case study

Authorization and authentication

Custom authorization policy

Conclusion

10. Implementing MicroServices with Web APIs

Introduction

Structure

Objectives

Implementing MicroServices with WebAPIs

Asynchronous communication

RabbitMQ

MicroServices scalability

Horizontal scalability

Vertical scalability

Orchestrators

Most used architectural patterns

Backend for frontend

Command Query Responsibility Segregation

Domain Driven Design

Case study

Conclusion

11. CI/CD with Docker and Azure DevOps

Introduction

Structure

Objectives

Overview

Docker

Docker containers

Container images

Docker images

Container images X Docker images

Docker advantages for your apps

Understanding the Dockerfile

Dockerfile for multi-stage builds

Docker commands

Azure DevOps

Continuous integration

Benefits of continuous integration

Continuous deployment

Benefits of continuous deployment

Case study

Creating the Dockerfile

Creating the Docker image

Run the image

Applying continuous integration

Applying continuous deployment

Conclusion

12. Building Multi-platform Apps with .NET MAUI and Blazor Hybrid

Introduction

Structure

Objectives

.NET MAUI overview

Differences between Blazor X Blazor Hybrid

Case study with step-by-step implementation

Creating the .NET MAUI project

Using Blazor Hybrid UI from Desktop client

Using Blazor Hybrid UI from mobile client

Conclusion

13. Windows UI Library: Crafting Native Windows Experience

Introduction

Structure

Objectives

Windows UI Library Introduction

Fluent Design System

Key principles of Fluent Design System

Applications of the Fluent Design System

Case study with step-by-step implementation

Creating the project

Designing the user interface

Implementing the cache

Data transfer between pages

Conclusion

14. Unit Testing and Debugging

Introduction

Structure

Objectives

Unit testing with xUnit

Making usage of mocks

Mastering debugging

Applying xUnit and mocks

Conclusion

Index

CHAPTER 1

Introduction to Visual Studio 2022

Introduction

This chapter explores the latest features and improvements in Microsoft's interactive development environment. The new version of Visual Studio provides developers with better productivity and enhanced collaboration capabilities. The chapter covers the significant changes, including the introduction of the 64-bit architecture, improved performance, and better integration with Azure services.

This chapter highlights the significant updates of the IDE's user interface, which includes an updated start window, a redesigned search experience, and a refreshed iconography. This chapter also covers the enhanced code analysis capabilities, which include code suggestion, auto-correction, and intelligent code completion. Additionally, the chapter delves into new debugging and testing features, including live unit testing, snapshot debugging, and time travel debugging.

Structure

This chapter covers the following topics:

- Significant changes from Visual Studio 2019
- Live unit testing
- Snapshot debugging
- Time travelling debugging

Objectives

At the end of this chapter, readers will have a solid understanding of the main changes and new features introduced in Visual Studio 2022, including live unit testing, snapshot debugging, and time travelling debugging. They will be equipped with the knowledge necessary to leverage these tools effectively in their C# development workflows for cloud, web, and desktop applications.

Significant changes from Visual Studio 2019

Visual Studio 2022 brings significant improvements and enhancements compared to Visual Studio 2019. These include a more modern and intuitive user interface with refreshed visual themes, simplified toolbars, and menus. The performance has been optimized, offering faster operations such as build acceleration for .NET SDK style projects, faster "Find in Files" functionality, and improved Git tooling speed. Visual Studio 2022 is now a 64-bit application, allowing for better utilization of system resources. The introduction of new features like Hot Reload for code files, enhanced Commit Graph, and improved support for multiple repositories enhances the overall development experience. Additionally, there are advancements in the Razor and C# experiences, including improved code navigation, smarter IntelliCode suggestions, and better code formatting. With these changes, Visual Studio 2022 empowers developers with a more efficient, productive, and enjoyable development environment.

Performance improvements

Visual Studio 2022 introduces a range of performance improvements that enhance the development experience for developers. These improvements target various areas of the IDE,

including build acceleration for .NET SDK style projects, external sources de-compilation, the Threads Window, Quick Add Item, code coverage, and Razor & C# experiences.

Let's delve into each of these enhancements:

- **Build acceleration for .NET SDK style projects:** Visual Studio 2022 incorporates optimizations for faster build times, specifically for .NET SDK style projects. By improving the build process up to 80%, developers experience reduced waiting times, allowing for quicker iterations and increased productivity.
- **External sources De-compilation:** This improvement enables developers to navigate and debug through code even when the source files are external 10x faster than Visual Studio 2019.
- **Find in files:** With enhanced search indexing, parallel processing, smarter search algorithms, search result previews, and advanced filtering options, developers can locate the desired information within their codebase 3x faster, reducing time spent on searching and improving overall productivity.
- **Git tooling:** Visual Studio 2022 introduces a new feature called the Commit Graph, which enhances the visualization and navigation of Git commits within the IDE. The Commit Graph in Visual Studio 2022 demonstrates a substantial improvement in performance, with an average loading time for branch history reduced by 70%.
- **Threads window:** The Threads Window in Visual Studio 2022 has undergone performance improvements, resulting in a more responsive and efficient debugging for applications with many threads. Developers can now effortlessly analyze and manage threads, gaining deeper insights into multi-threaded applications and improving overall debugging productivity.
- **Quick add item:** Visual Studio 2022 introduces the “New Item” menu command, which accelerates the process of adding new items to projects. With this enhancement, developers can swiftly add multiple files and folders streamlining the project creation and modification process.
- **Code coverage:** Code coverage analysis in Visual Studio 2022 has been optimized for 35% faster code coverage tests. Developers can now measure the code coverage of their tests more efficiently, allowing them to identify areas of code that lack test coverage and improve overall code quality.

- **Razor and C# experience:** Visual Studio 2022 brings support for code actions, including some helpful shortcuts that are vital for web development. These enhancements enhance the responsiveness and responsiveness of code editing, navigation, and IntelliSense features, enabling developers to write, modify, and navigate code more smoothly.

With these performance improvements in Visual Studio 2022, developers can expect a more efficient and responsive development environment. The build acceleration for .NET SDK style projects, external sources de-compilation, improvements to the Threads Window, Quick Add Item feature, enhanced code coverage analysis, and optimized Razor and C# experiences collectively enhance productivity and streamline the development process, enabling developers to deliver high-quality applications more efficiently.

64-bit support

One of the significant enhancements in Visual Studio 2022 is its transition to a 64-bit application. This shift from a 32-bit to a 64-bit architecture brings several advantages and benefits to developers.

Let's explore the significance of Visual Studio 2022 being a 64-bit application:

- **Increased memory access:** As a 64-bit application, Visual Studio 2022 can take advantage of the expanded memory address space provided by 64-bit systems. This allows the IDE to access larger amounts of memory, enabling developers to work with larger and more complex projects without facing memory limitations or performance issues. Developers may now work with solutions containing more projects, capitalizing on the extended memory support offered by the 64-bit architecture.
- **Compatibility with 64-bit Tools and Libraries:** Being a 64-bit application, Visual Studio 2022 seamlessly integrates with other 64-bit tools, libraries, and components, ensuring compatibility and interoperability. Developers can take full advantage of the advancements and optimizations provided by 64-bit technologies in the development ecosystem.
- **Future-Proofing:** The transition to a 64-bit application positions Visual Studio 2022 for future growth and scalability. It aligns with the industry's shift towards 64-bit computing and ensures that the IDE can accommodate the evolving

demands of modern development environments and technologies.

This transition empowers developers with a more robust and capable development environment, enabling them to tackle complex projects, streamline workflows, and deliver high-quality applications with greater efficiency.

Smarter IntelliCode

Program Synthesis using Examples (PROSE) technology is an automated program synthesis framework developed by Microsoft Research. It enables developers to generate code automatically based on input-output examples, allowing for rapid prototyping and development. PROSE leverages machine learning and programming language techniques to understand and generalize patterns from provided examples, reducing the manual effort required in writing code. It has been applied to various domains, including data wrangling, API usage, and code refactoring. PROSE aims to enhance developer productivity by automating repetitive coding tasks through the power of example-based program synthesis.

IntelliCode in Visual Studio 2022 actively analyzes your code changes as you type, leveraging PROSE technology to identify common patterns in manual code modifications. By recognizing how developers typically perform automatable code changes, IntelliCode offers helpful suggestions to streamline your coding process. When you're in the middle of a code fix or refactoring, IntelliCode presents suggestions from the completions list, allowing you to effortlessly complete the code change with a single click. This intelligent assistance provided by IntelliCode optimizes your coding experience, enabling you to work more efficiently and with greater accuracy.

Razor editor improvements

The new Razor editor introduces additional code fixes and refactoring, including the popular "Add missing usings" refactoring. It also offers Razor-specific refactoring like "Extract block to code behind" for easy extraction of code blocks to separate files.

Navigation support is improved with the "Go to Definition" feature for components, allowing quick navigation within files to

understand code better.

Default colors in the new Razor editor have been updated, removing the code background highlight for better readability and reduced visual clutter.

The new Razor editor supports the latest compiler features, providing smarter Razor syntax completions such as "< text >" completion and auto-complete. Diagnostics are streamlined to show only the most important issues and maintain the intended fidelity of compiler-generated diagnostics.

Razor now fully integrates with Visual Studio Live Share, facilitating remote collaboration and code sharing among developers, supporting a shared context for efficient co-programming.

Hot reload with new capabilities

The introduction of the new Hot Reload technology brings seamless code file updates alongside XAML Hot Reload, catering to applications utilizing XAML for their UI. Whether you're working with XAML or .NET, Hot Reload is available, providing a powerful development experience. Hot Reload seamlessly integrates with familiar debugging capabilities like breakpoints, **'edit and continue' (EnC)**, and other essential features.

In this release, Hot Reload is compatible with a wide range of application types. It works effortlessly with XAML-powered applications such as WPF, .NET MAUI and WinUI 3, as well as other frameworks like Windows Forms, ASP.NET web apps, Blazor Server, Console apps, and more. Essentially, wherever a modern .NET runtime is used in conjunction with the Visual Studio debugger, Hot Reload can significantly enhance your development workflow.

Multi-repository support

Visual Studio 2022 now offers support for multiple repositories, allowing you to work with up to 10 active Git repositories simultaneously. This feature enables seamless collaboration and efficient management of solutions that span across multiple repositories. You can perform Git operations across various repositories concurrently, streamlining your workflow.

For instance, in a large-scale web project, you might have separate repositories for the frontend, API, database, documentation, and various libraries or dependencies. Previously, managing work across these repositories required opening multiple instances of Visual Studio. However, with the multi-repository support, you can now handle, view, and debug all these repositories within a single instance of Visual Studio. This capability enhances productivity by eliminating the need to switch between different IDE instances and allows for a consolidated and cohesive development experience across multiple repositories.

Interactive staging support

Line-staging empowers you to divide your modified lines of code into separate commits, offering enhanced flexibility in managing your changes. This feature can be leveraged to review your alterations before finalizing the commits. By selectively staging specific lines or sections of code, you can mark them as reviewed, ensuring thorough scrutiny before committing. Once you are satisfied with the staged changes, you can proceed to commit them, providing a controlled and organized approach to incorporating your modifications into the codebase.

Interface improvements and customization

Visual Studio 2022 introduces a fresh and modern look & feel, enhancing the overall developer experience. The new design brings a sleek and intuitive interface, providing a more streamlined and visually appealing environment for software development, as we can see from the picture below:

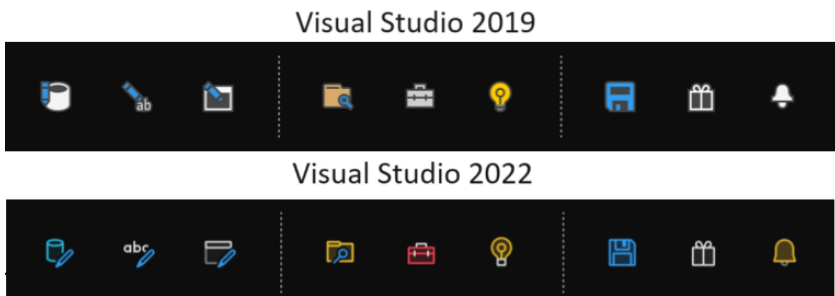


Figure 1.1: Visual Studio 2022 icons comparison with Visual Studio 2019.

Here are the key highlights of the new look and feel in Visual Studio 2022:

- **Refreshed visual theme:** The IDE features a refined visual theme that aligns with contemporary design principles. The interface incorporates updated icons, cleaner layouts, and improved typography, resulting in a more visually pleasing and cohesive experience.
- **Simplified toolbar and menus:** The toolbar and menus have been streamlined to prioritize commonly used commands and reduce clutter. This simplification enhances navigation and allows for quicker access to essential tools and features, promoting a more efficient workflow.
- **Enhanced iconography:** Visual Studio 2022 introduces a new set of modern icons that are visually consistent and easily recognizable. The redesigned icons provide a clearer representation of commands and functionalities, making it easier for developers to identify and use the desired features.
- **Improved editor experience:** The code editor in Visual Studio 2022 benefits from a refined visual design, offering better readability and an enhanced focus on code content. Syntax highlighting, code suggestions, and other editing features have been optimized for improved visibility and ease of use.
- **Customizable window layouts:** Visual Studio 2022 empowers developers to personalize their workspace by customizing window layouts. You can arrange tool windows, panes, and panels according to your preferences, creating a workspace that suits your specific needs and enhances productivity.
- **Sync your custom settings:** With the capability to sync your settings, you can maintain productivity seamlessly, regardless of whether you're working from home or the office. Visual Studio offers robust features to sync the settings that matter to you, allowing you to store them securely in the cloud. This means you can access your personalized settings and configurations from any location, enabling you to work at your best and maximize efficiency wherever you find most convenient. Whether you switch between different machines or work remotely, the ability to sync settings ensures a consistent and optimized development environment that supports your productivity no matter where you are.

- **Improved dark theme and custom themes:** The new dark theme in Visual Studio 2022 brings significant enhancements to contrast, accent color, and accessibility, ensuring a better visual experience for most users. Furthermore, the Visual Studio marketplace offers a wide array of custom themes, providing you with the flexibility to choose the theme that best suits your preferences and optimizes your workflow. Whether you opt for the improved default dark theme or decide to customize Visual Studio with a theme of your choice, we strive to make your Visual Studio experience tailored to your needs and preferences.

The new look and feel in Visual Studio 2022 introduce a modern and visually appealing interface, enhancing the overall user experience. Visual Studio 2022 provides a fresh and enjoyable environment for developers to create their software solutions.

Live unit testing

As you make modifications to your code, Live Unit Testing offers valuable insights into the impact of your changes on existing tests, ensuring that newly added code is covered by one or more relevant tests in real time. This serves as a gentle reminder to prioritize unit testing during bug fixes and feature additions, promoting robust and reliable development practices.

Using the live unit testing

The following steps outline both the limitations of Live Unit Testing and provide guidance on how to set it up effectively.

To enable **Live Unit Testing** go to **Test | Live Unit Testing | Start**. The shortcut is **Ctrl E + L**:

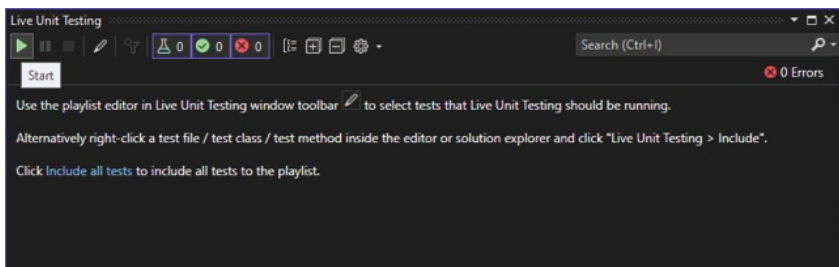


Figure 1.2: Live Unit Testing Window

Then you can configure according to your needs:

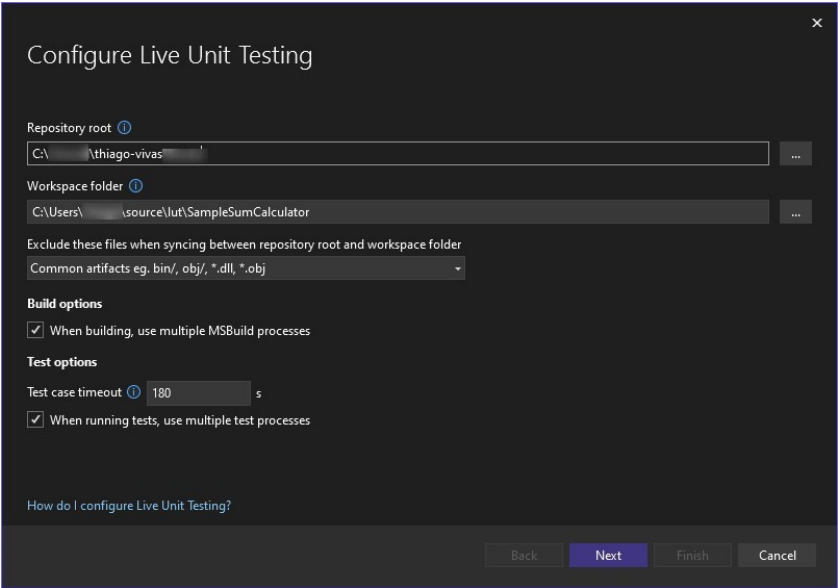


Figure 1.3: Live Unit Testing Configuration Window

Example of Live Unit Tests in execution:

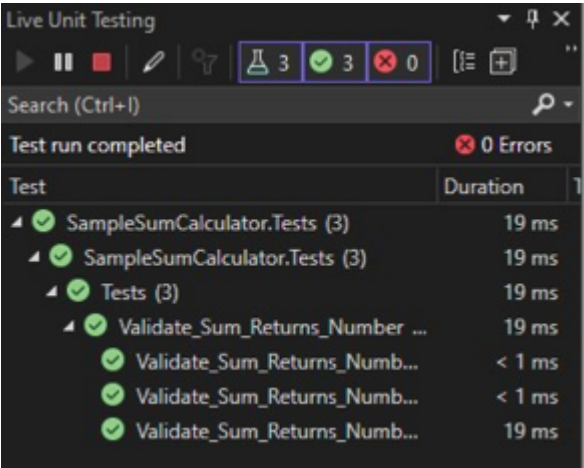


Figure 1.4: Live Unit Testing Window with passed Unit Tests

Supported testing frameworks

By the publication of this book, **Live Unit Testing** is available only in the Visual Studio Enterprise version and on the following:

- xUnit.Net
 - Visual Studio Adapter minimum version:
 - xunit.runner.visualstudio version 2.2.0-beta3-build1187
 - Framework minimum version:
 - xunit 1.9.2
- NUnit
 - Visual Studio Adapter minimum version:
 - NUnit3TestAdapter version 3.5.1
 - Framework minimum version:
 - NUnit version 3.5.0
- MSTest
 - Visual Studio Adapter minimum version:
 - MSTest.TestAdapter 1.1.4-preview
 - Framework minimum version:
 - MSTest.TestFramework 1.0.5-preview

Exclude and include test projects and test methods

By right clicking on your **Unit Tests** you can include or exclude them from the **Live Unit Testing**:

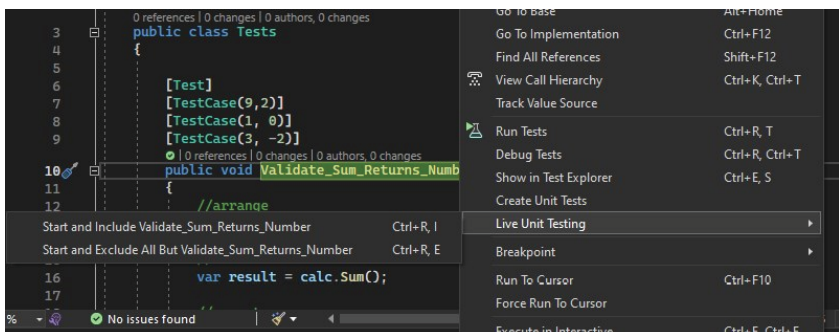


Figure 1.5: Window to include or exclude test methods

You can also exclude **Unit Tests** programmatically:

- To exclude single unit tests:

- ☐ NUnit

- ☐

```
[Category("SkipWhenLiveUnitTesting")]
```

- ☐ xUnit

- ☐

```
[Trait("Category",  
"SkipWhenLiveUnitTesting")]
```

- ☐ MSTest

- ☐

```
[TestCategory("SkipWhenLiveUnitTesting")]
```

- To exclude the entire assembly

- ☐ NUnit

- ☐

```
[assembly:  
Category("SkipWhenLiveUnitTesting")]
```

- ☐ xUnit

- ☐

```
[assembly: AssemblyTrait("Category",  
"SkipWhenLiveUnitTesting")]
```

- ☐ MSTest

- ☐

```
[assembly:  
TestCategory("SkipWhenLiveUnitTesting")]
```

You can also include or exclude entire Unit Testing Projects by right clicking on its project:

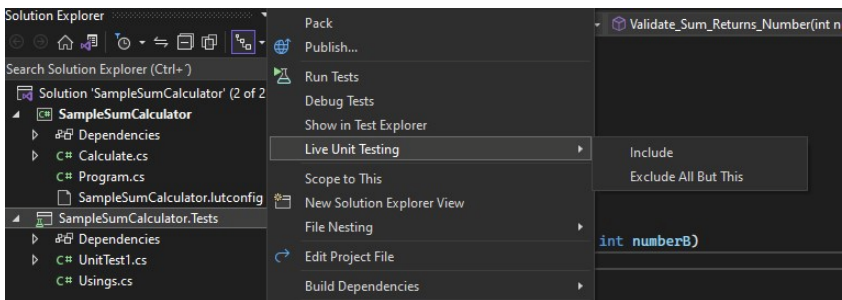


Figure 1.6: Menu option to include or exclude Unit Test Projects

Configuring according to your needs

From the **Tools | Options** Menu you can customize your Live Unit Test according to your needs:

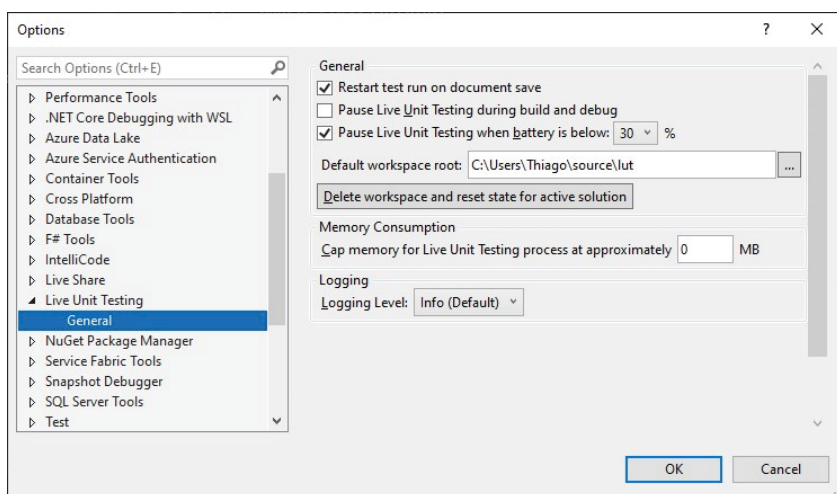


Figure 1.7: Options window to configure Live Unit Testing

Snapshot debugging

The Snapshot Debugger available in Application Insights empowers you to effortlessly gather a debug snapshot of your web application in case of an exception. This invaluable snapshot unveils the precise state of your source code and variables at the exact moment the exception occurred, giving you the possibility to simulate the scenario that throw the exception.

The Snapshot Debugger in Application Insights main functionalities:

- Collection of snapshots for your most frequently thrown exceptions.
- Monitoring of system-generated logs from your web app.
- Provision of essential information necessary for diagnosing, simulating, and resolving issues within your production environment.

Using snapshot debugging

The following steps elucidate the limitations of Snapshot debugging and offer comprehensive instructions on how to set it up seamlessly.

1. Install the Nuget package in your application

Microsoft.ApplicationInsights.SnapshotCollector

2. Configure your Application Insights in your application

a. For ASP.NET Applications

i. Update your **ApplicationInsights.config**. This is the default configuration:

- `<TelemetryProcessors>`
- `<Add`
`Type="Microsoft.ApplicationInsights.Snap`
`Microsoft.ApplicationInsights.SnapshotCol`
- `<!-- The default is true, but you can disable`
`Snapshot Debugging by setting it to false -->`
- `<IsEnabled>true</IsEnabled>`
- `<!-- Snapshot Debugging is usually disabled`
`in developer mode, but you can enable it by`
`setting this to true. -->`
- `<!-- DeveloperMode is a property on the`
`active TelemetryChannel. -->`
- `<IsEnabledInDeveloperMode>>false</`
`IsEnabledInDeveloperMode>`
- `<!-- How many times we need to see an`
`exception before we ask for snapshots. -->`
- `<ThresholdForSnapshotting>1</`
`ThresholdForSnapshotting>`
- `<!-- The maximum number of examples we`
`create for a single problem. -->`
- `<MaximumSnapshotsRequired>3</`
`MaximumSnapshotsRequired>`
- `<!-- The maximum number of problems that`
`we can be tracking at any time. -->`
- `<MaximumCollectionPlanSize>50</`

```
MaximumCollectionPlanSize>
```

- <!-- How often we reconnect to the stamp.
The default value is 15 minutes.-->

```
<ReconnectInterval>00:15:00</  
ReconnectInterval>
```

- <!-- How often to reset problem counters. -->

```
<ProblemCounterResetInterval>1.00:00:00</  
ProblemCounterResetInterval>
```

- <!-- The maximum number of snapshots
allowed in ten minutes. The default value is 1. -->

```
<SnapshotsPerTenMinutesLimit>3</  
SnapshotsPerTenMinutesLimit>
```

- <!-- The maximum number of snapshots
allowed per day. -->

```
<SnapshotsPerDayLimit>30</  
SnapshotsPerDayLimit>
```

- <!-- Whether or not to collect snapshot in low
IO priority thread. The default value is true. -->

```
<SnapshotInLowPriorityThread>true</  
SnapshotInLowPriorityThread>
```

- <!-- Agree to send anonymous data to
Microsoft to make this product better. -->

```
<ProvideAnonymousTelemetry>true</  
ProvideAnonymousTelemetry>
```

- <!-- The limit on the number of failed
requests to request snapshots before the
telemetry processor is disabled. -->

```
<FailedRequestLimit>3</  
FailedRequestLimit>
```

- </Add>

○ `</TelemetryProcessors>`

ae. For ASP.NET Core Applications

i. Create a new class named

SnapshotCollectorTelemetryProcessorFactory

○ `using`
`Microsoft.ApplicationInsights.AspNetCore;`

○ `using`
`Microsoft.ApplicationInsights.Extensibility;`

○ `using`
`Microsoft.ApplicationInsights.SnapshotCollector;`

○ `using` `Microsoft.Extensions.Options;`

○ `internal class`
`SnapshotCollectorTelemetryProcessorFactory`
`: ITelemetryProcessorFactory`

○ `{`

○ `private readonly` `IServiceProvider`
`_serviceProvider;`

○ `public`
`SnapshotCollectorTelemetryProcessorFactory` (`IServiceProvider` `serviceProvider`) =>

○ `_serviceProvider = serviceProvider;`

○ `public` `ITelemetryProcessor`
`Create` (`ITelemetryProcessor` `next`)

○ `{`

○ `IOptions<SnapshotCollectorConfiguration>`
`snapshotConfigurationOptions =`
`_serviceProvider.GetRequiredService<IOptions<SnapshotCollectorConfiguration>>();`

○ `return new`
`SnapshotCollectorTelemetryProcessor` (`next`,
`configuration:`
`snapshotConfigurationOptions.Value`);

○ `}`

○ `}`

xvii. Configure the Dependency Injection

- `using`
`Microsoft.ApplicationInsights.AspNetCore;`
- `using`
`Microsoft.ApplicationInsights.SnapshotCollector;`
- `var` builder =
`WebApplication.CreateBuilder(args);`
- `builder.Services.AddApplicationInsightsTelemetry();`
- `builder.Services.AddSnapshotCollector(config =>`
`builder.Configuration.Bind(nameof(SnapshotCollectorConfiguration), config));`
- `builder.Services.AddSingleton<ITelemetryProcessorFactory>(s => new`
`SnapshotCollectorTelemetryProcessorFactory(s));`

xxiv. If you need any customization then you need to update your **appsettings.json**. Following we have a default configuration:

1. {
2. "SnapshotCollectorConfiguration": {
3. "IsEnabledInDeveloperMode": false,
4. "ThresholdForSnapshotting": 1,
5. "MaximumSnapshotsRequired": 3,
6. "MaximumCollectionPlanSize": 50,
7. "ReconnectInterval": "00:15:00",
8. "ProblemCounterResetInterval": "1.00:00:00",
9. "SnapshotsPerTenMinutesLimit": 1,
10. "SnapshotsPerDayLimit": 30,
11. "SnapshotInLowPriorityThread": true,
12. "ProvideAnonymousTelemetry": true,
13. "FailedRequestLimit": 3
14. }
15. }

Supported frameworks and environments

By the time of the publication of this book, the following frameworks and environments support Snapshot debugging:

- Frameworks
 - .NET Framework 4.6.2 and newer versions.
 - .NET 6.0 or later on Windows.
- Environments
 - Azure App Service
 - Azure Functions
 - Azure Cloud Services running OS family 4 or later
 - Azure Service Fabric running on Windows Server 2012 R2 or later
 - Azure Virtual Machines and Azure Virtual Machine Scale Sets running Windows Server 2012 R2 or later
 - On-premises virtual or physical machines running Windows Server 2012 R2 or later or Windows 8.1 or later

Required permissions and limitations

The following are the required permissions and limitations:

- Snapshots are protected by **Azure Role-Based Access Control (RBAC)**. To inspect a snapshot, you must be added in the Application Insights Snapshot debugger role.
- Snapshots might contain sensitive data.
- Snapshots are stored in the same region as your Application Insights resource.
- Snapshots are stored for 15 days by default. An increase can be requested.
- Snapshots requires symbol files to be available in the server to decode variables and enable debugging in Visual Studio.
- Local variables may not be available when having optimized builds.

Time Travelling Debugging

Introducing **Time Travel Debugging (TTD)**, a groundbreaking

capability that grants you the authority to record and replay code execution within Visual Studio, directly from your production environment. TTD goes beyond traditional debugging methods by enabling seamless navigation through time, akin to conducting "inner loop" debugging locally. This means you can effortlessly move both forward and backward in the execution timeline, while retaining access to essential debugging features like locals and the call stack.

Conventional debuggers, even with advanced tools like IntelliTrace, typically allow halting at specific breakpoints and only moving forward in time. However, TTD offers a significant advantage over other methods such as snapshots, logging, or crash dump files. These traditional techniques often lack the precise details of the execution path leading up to the final failure or bug. TTD fills this gap by capturing the exact execution path, providing an unprecedented level of insight into the context surrounding the issue, surpassing the limitations of snapshots and other approaches.

Using time travelling debug

The following steps shed light on the limitations of time-traveling debugging and provide a comprehensive guide on how to set it up effortlessly.

Recording a snapshot

When it comes to time-traveling debugging, recording snapshots is a fundamental aspect of the process. Time-traveling debugging allows developers to replay and analyze the execution of their code, stepping forward and backward through time to identify and debug issues. Recording snapshots enables the capture and preservation of program state at different points in time, facilitating the exploration of program behavior and the identification of bugs.

From Debug |Attach Snapshot debugger you can attach the Snapshot debugger:

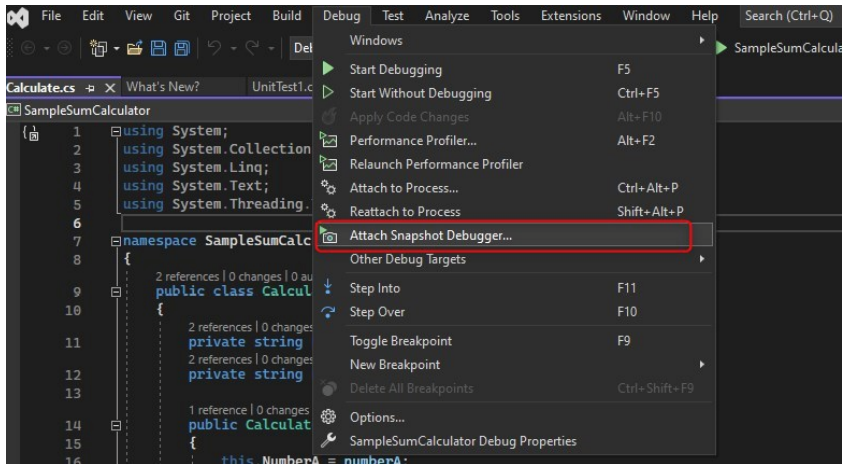


Figure 1.8: Menu to attach the Snapshot debugger

Select your Virtual Machine and check both options. Do not forget about the Virtual Machine specificities.

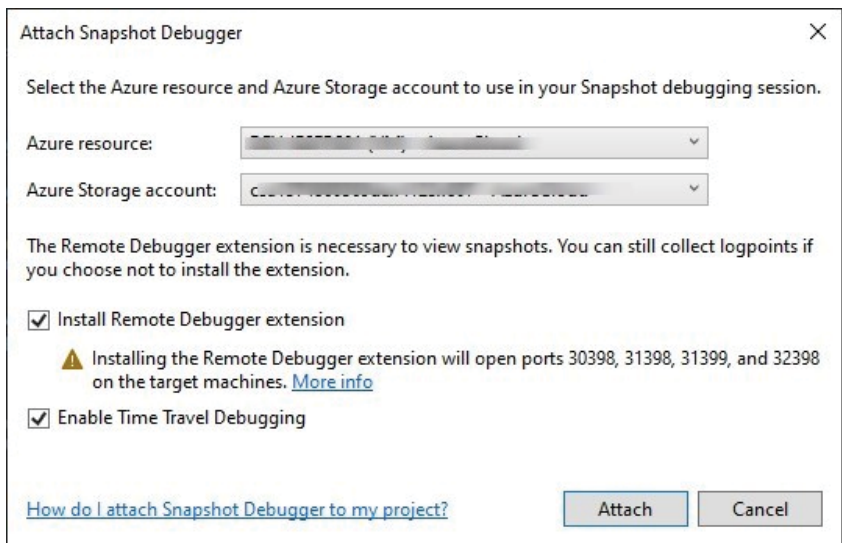


Figure 1.9: Window with the Attach Snapshot debugger settings

Select a breakpoint and in its breakpoint settings, you have to check the "Collect a time travel trace to the end of this method" option.

You will be able to see the snapshots recorded at the Diagnostic

Tools window.

Viewing the snapshots

Once the snapshots are recorded you can play it from the Diagnostic Tools window and use the debugger as normal.

Time travelling debug limitation

By the time of this book publication, those are the requirements for running the TTD:

- Visual Studio Enterprise
- Azure Virtual Machine Running Windows OS with ASP.NET 4.8+
- AMD64 Web Apps

Conclusion

In this chapter we provided an in-depth introduction to Visual Studio 2022, focusing on the main changes from its predecessor, Visual Studio 2019, as well as the exciting new features it brings to the table. We explored the concepts of live unit testing, snapshot debugging, and time travelling debugging, highlighting their significance in the development process.

Visual Studio 2022 brings a plethora of improvements and enhancements that aim to streamline the development experience and boost productivity. The changes in the user interface, code editing capabilities, and debugging tools contribute to creating a more efficient and enjoyable environment for developers working on C# applications.

In the upcoming chapter, we will delve into the exciting realm of C# 11, exploring a wide range of new features and enhancements introduced in the latest version of the language.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 2

What is New in C# 12

Introduction

Welcome to the exciting world of C# 11 and C# 12! In this chapter, we will discuss the latest enhancements and features introduced in these versions of the C# programming language. From improved string handling to enhanced pattern matching and method group conversions, C# continues to evolve, offering developers powerful tools to build robust and efficient applications.

We will start by exploring the main changes introduced in C# 11, including raw string literals, generic math support, and file-local types. Then, we will dive into the primary constructors, collection expressions, and other exciting additions brought by C# 12. Along the way, we will discuss how these features enhance productivity, readability, and performance, empowering developers to write cleaner and more maintainable code.

Whether you are a seasoned C# developer or just starting your journey with the language, this chapter will provide valuable insights into the latest advancements in C# development. Let us discuss the cutting-edge features that C# 11 and C# 12 have to offer!

Structure

This chapter covers the following topics:

- C# 11 updates
 - Raw string literals
 - Generic math support
 - Generic attributes
 - Unicode Transformation
 - Newlines in string interpolation expressions
 - List patterns
 - File-local types
 - Required members
 - Auto-default structs
 - Pattern match `Span<char>` on a constant string
 - Extended nameof scope
 - Numeric IntPtr and UIntPtr
 - ref fields and scoped ref
 - Improved method group conversion to delegate
 - Warning wave 7
- C# 12 updates
 - Primary constructors
 - Collection expressions
 - Inline arrays
 - Optional parameters in lambda expressions
 - ref readonly parameters
 - Alias any type

Objectives

In this chapter, our primary goal is to provide a comprehensive overview of the key features introduced in C# 11 and C# 12, empowering developers to leverage the latest advancements in the language effectively. We aim to familiarize readers with the main

changes from C# 10, such as raw string literals, generic math support, and file-local types, before delving into the enhancements introduced in subsequent versions.

Through clear explanations and practical examples, we strive to help developers understand the benefits and applications of each new feature, enabling them to write cleaner, more efficient, and maintainable code. By the end of this chapter, readers will have gained a solid understanding of the advancements in C# 11 and C# 12, equipping them with the knowledge and skills to leverage these features in their own projects effectively.

C# 11 updates

This section will discuss the significant updates introduced in C# 11, enhancing developer capabilities and code efficiency. From the introduction of primary constructors to the inline arrays, C# 11 introduces a variety of features aimed at streamlining development processes and improving code readability. Developers will explore primary constructors, collection expressions, inline arrays, optional parameters in lambda expressions, ref readonly parameters, aliasing any type, and the introduction of inline arrays. Understanding these changes empowers developers to leverage the latest tools and techniques to create robust, efficient, and maintainable code in their C# projects.

Raw string literals

Raw string literals introduce a new format for representing strings. They can encompass arbitrary text, including whitespace, new lines, embedded quotes, and other special characters without the need for escape sequences. The initiation of a raw string literal involves at least three double-quote (""") characters, and its conclusion mirrors the same count of double-quote characters. Typically, a raw string literal is initialized using three double quotes on a single line to commence the string, and three double quotes on a separate line to conclude it. Notably, the newlines following the opening quote and preceding the closing quote are not considered part of the final content, as we can see in the following example:

```
1. string multiLineText = """
```

```
2.   This represents a multiline text.
```


3. It spans across various lines.
4. Some lines have varying levels of indentation.
5. A few start at the first column.
6. Quoted text is included within some lines.
7. `"";`

The leading whitespace to the left of the closing double quotes is excluded from the resulting string literal. Combining raw string literals with string interpolation allows the inclusion of braces in the output text. The number of consecutive `$` characters signifies the beginning and end of the interpolation, determining the extent of braces in the output, as we can see below:

1. `var locationInfo = $$""`
2. Your current location is at `{{{Longitude}}},
{{{Latitude}}}`
3. `"";`

In the previous illustration, it is indicated that two consecutive braces mark the initiation and conclusion of an interpolation. The third set of opening and closing braces, when repeated, becomes part of the resulting output string.

Generic math support

C# 11 introduces several language features that enhance support for generic math:

- **Static virtual members in interfaces:** You can now incorporate static abstract or static virtual members in interfaces, allowing the definition of interfaces with overloadable operators, static members, and properties. This facilitates the use of mathematical operators in generic types. For instance, by implementing the `System.IAdditionOperators<TSelf, TOther, TResult>` interface, a type that can support the addition operator. Additional interfaces cover various mathematical operations or well-defined values. The syntax details can be explored in the interfaces article. Typically, interfaces with static virtual methods are generic and impose constraints on the type parameter to implement the declared interface.
- **Generic math requirements:** The advent of generic math introduces specific requirements for the language.

- **Unsigned right shift operator:** C# 11 introduces the `>>>` operator, allowing an unsigned right shift directly. Previously, achieving an unsigned right shift necessitated casting a signed integer type to an unsigned type, performing the shift, and then casting the result back to a signed type.
- **Relaxed shift operator requirements:** The requirement that the second operand must be an `int` or implicitly convertible to `int` is removed in C# 11. This change enables types implementing generic math interfaces to be utilized in these shift operator locations.
- **Checked and unchecked user-defined operators:** Developers now have the ability to define both checked and unchecked arithmetic operators. The compiler automatically generates calls to the appropriate variant based on the current context. Further details on checked operators can be explored in the article on Arithmetic operators.

These enhancements collectively contribute to a more versatile and expressive approach to generic math operations in C# 11, as we can see in the code below:

```

1. public static TResult CalculateSum<T,
   TResult>(IEnumerable<T> data)
2.     where T : INumber<T>
3.     where TResult : INumber<TResult>
4. {
5.     TResult result = TResult.Zero;
6.
7.     foreach (var value in data)
8.     {
9.         result += TResult.Create(value);
10.    }
11.
12.    return result;
13. }
14.
15. public static TResult CalculateAverage<T,
   TResult>(IEnumerable<T>
16. data)
17.     where T : INumber<T>

```

```

18.     where TResult : INumber<TResult>
19. {
20.     TResult sum = CalculateSum<T,
    TResult>(data);
21.     return TResult.Create(sum) /
    TResult.Create(data.Count());
22. }
23.
24. public static TResult
    CalculateStandardDeviation<T, TResult>
25. (IEnumerable<T> data)
26.     where T : INumber<T>
27.     where TResult : IFloatingPoint<TResult>
28. {
29.     TResult standardDeviation = TResult.Zero;
30.
31.     if (data.Any())
32.     {
33.         TResult average =
    CalculateAverage<T, TResult>(data);
34.         TResult sum = CalculateSum<T,
    TResult>(data.Select((value) => {
35.             var deviation =
    TResult.Create(value) - average;
36.             return deviation * deviation;
37.         }));
38.         standardDeviation = TResult.Sqrt(sum / TResult
39. .Create(data.Count() - 1));
40.     }
41.     return standardDeviation;
42. }

```

Generic attributes

You have the option to define a generic class with a base class of **System.Attribute**. This feature simplifies the syntax for attributes that necessitate a **System.Type** parameter. In the past, you would

have had to design an attribute with a constructor parameter requiring a `Type`.

The type arguments must adhere to the constraints imposed by the `typeof` operator. Types requiring metadata annotations are prohibited. For instance, the following types are not permissible as type parameters:

- `dynamic`
- `string?` (or any nullable reference type)
- `(int X, int Y)` (or any other tuple types using C# tuple syntax)

These types incorporate annotations describing the type and are not directly represented in metadata. In such cases, it is recommended to use the underlying types:

- Use `object` in place of `dynamic`
- Prefer `string` over `string?`
- Substitute `(int X, int Y)` with `ValueTuple<int, int>`

This is how we had to declare a generic attribute before C# 11:

```
1. public class SampleAttribute : Attribute
2. {
3.     public SampleAttribute(Type t) =>
        ParamType = t;
4.
5.     public Type ParamType { get; }
6. }
```

This is how we would use this generic attribute:

```
1. public class Class1
2. {
3.
4.     [SampleAttribute(typeof(string))]
5.     public string SampleMethod() =>
        default;
6. }
```

After those changes in C# 11 this is how we declare a generic attribute:

```
1. public class SampleAttribute<T> : Attribute {
```

```
}
```

This is how we use the generic attribute in C# 11:

```
1. public class Class1
2. {
3.     [SampleAttribute<string>() ]
4.     public string Method() => default;
5. }
```

Unicode Transformation Format-8 string literals

You have the option to use the `u8` suffix on a string literal to indicate Unicode Transformation Format, or UTF-8 character encoding. If your application requires UTF-8 strings, especially for scenarios like HTTP string constants or similar text protocols, leveraging this feature can streamline the process of creating UTF-8 strings.

Thus, we attain optimal performance without the complexity of additional code, no necessity to declare a field for holding the UTF-8 representation.

By appending the `u8` suffix to a string literal, the resultant type becomes a `ReadOnlySpan<byte>`. The compiler handles the emission of string data into the DLL in a manner identical to the earlier example involving a static property returning a `ReadOnlySpan<byte>`. Consequently, we gain the performance advantages of this approach without any associated inconveniences.

With the changes in C# we can write this:

```
1. ReadOnlySpan<byte> SampleStringLiteral =
   "Sample"u8;
```

Default values for optional parameters cannot be set using UTF-8 string literals as they are runtime constants rather than compile-time constants. Furthermore, string interpolation with UTF-8 string literals is not allowed, as they cannot be used in conjunction with the `$` token and the `u8` suffix within the same string expression.

It is crucial to understand that while `ReadOnlySpan<byte>` or `byte[]` types are compile-time enforced, UTF-8 strings lack this compile-time enforcement. Consequently, attempting to use UTF-8 strings as default parameters in functions will lead to a compilation


```

4.     score switch
5.     {
6.         > 90 => "Unlimited usage",
7.         > 80 => "General usage, with daily
            safety check",
8.         > 70 => "Issues must be addressed
            within 1 week",
9.         > 50 => "Issues must be addressed
            within 1 day",
10.        _ => "Issues must be addressed
            before continued use"
11.    };

```

While your perspective is valid, there are occasions where string interpolation remains preferable. Notably, with the introduction of C# 11, string interpolation has undergone performance enhancements, resulting in significantly lower execution time and memory consumption. We have a comparison of ways to concatenate strings below:

- The least efficient option among the mentioned string concatenation methods is `string.Concat()` (the overloads accepting objects, not strings), primarily because it consistently boxes structs and generates intermediate strings, as observed through decompilation using ILSpy.
- Both `+` and `string.Concat()` (the overloads accepting strings, not objects) exhibit slightly better performance characteristics. Although they utilize intermediate strings, they do not involve the boxing of structs.
- Moving on, `string.Format()` is generally more efficient compared to the previously mentioned methods. It does not necessarily box structs and avoids creating an intermediate string, especially when the structs implement `ISpanFormattable`, a performance-enhancing feature that was previously internal to .NET.
- The most efficient option, particularly with the release of .NET 6, is the usage of interpolated strings. They stand out as the superior choice because they neither box structs nor create intermediate strings for types implementing `ISpanFormattable`. With interpolated strings, the primary allocation is the returned string itself, making them a highly

optimized choice.

List patterns

List patterns broaden pattern matching to accommodate sequences of elements in a list or an array. For instance, the condition `sequence is [1, 2, 3]` evaluates to true when the sequence represents an array or a list containing three integers (1, 2, and 3). This allows matching elements using various patterns, such as constant, type, property, and relational patterns. The discard pattern `()` is employed to match any individual element, while the newly introduced range pattern `(..)` is utilized to match any sequence with zero or more elements.

With those changes, we can have something like this:

```
1. int[] values = { 2, 4, 6 };
2.
3. Console.WriteLine(values is [2, 4, 6]);
   // True
4. Console.WriteLine(values is [2, 4, 8]);
   // False
5. Console.WriteLine(values is [2, 4, 6, 8]);
   // False
6. Console.WriteLine(values is [0 or 2, <= 4,
   >= 6]); // True
```

As illustrated in the previous example, a list pattern is considered matched when each nested pattern aligns with the corresponding element in an input sequence. Within a list pattern, you have the flexibility to employ any pattern. If your goal is to match any element, you can utilize the discard pattern. Alternatively, if you also intend to capture the element, the var pattern can be employed, as demonstrated in the following example:

```
1. List<int> elements = new() { 7, 8, 9 };
2.
3. if (elements is [var initial, _, _])
4. {
5.     Console.WriteLine($"The lead element
6.                         in a three-item
   collection is {initial}.");
}
```


7. }
8. *// Output:*
9. *// The lead element in a three-item collection is 7.*

According to the documentation, list pattern matching in C# involves three distinct patterns: Discard Pattern, Range Pattern, and Var Pattern. Let us delve into each of these patterns to understand their significance:

The Discard Pattern proves useful when matching one or more elements from a sequence, provided we are aware of the sequence's length.

1. `int[] fibonacciSequence = { 1, 2, 3, 5, 8 };`
2. `bool matchingResult = false;`
- 3.
4. *// Matching result is false, length does not match*
5. `matchingResult = fibonacciSequence is [_,`
`_, 3, _];`
- 6.
7. *// Matching result is false, 3 is not at the same position*
8. `matchingResult = fibonacciSequence is [_,`
`_, _, 3, _];`
- 9.
10. *// Matching result is false, length matches, but 2 and 3 are not*
11. *// at the same positions*
12. `matchingResult = fibonacciSequence is [2,`
`_, 3, _, _];`
- 13.
14. *// Matching result is true, single element and its position*
15. *// and length match*
16. `matchingResult = fibonacciSequence is [1,`
`_, _, _, _];`
- 17.
18. *// Matching result is true, multiple elements and their positions*
19. *//and length match*

```
20. matchingResult = fibonacciSequence is [1,
    _, 3, _, _];
```

In the code above, you can observe the specification of elements for list pattern matching, with the constraint being the necessity to know the sequence's length for accurate comparison; otherwise, the comparison would invariably return false.

For scenarios where the length of the sequence to be compared is unknown, the Range Pattern becomes valuable. The use of two dots in this pattern indicates that any number of elements may exist in place of those two dots. It is important to note that the two dots can only be used once in the sequence:

```
1. int[] customFibonacci = { 1, 2, 3, 5, 8 };
2. bool matchingResult = false;
3.
4. // Matching result is true, as customFibonacci ends with
   element 8
5. matchingResult = customFibonacci is [...,
    8];
6.
7. // Matching result is false, 3 is not the second last element
8. matchingResult = customFibonacci is [..., 3,
    _];
9.
10. // Matching result is true, as sequence begins with 1 and ends
    with 8
11. matchingResult = customFibonacci is [1, ...,
    8];
12.
13. // Matching result is true, as 1 is the first element
14. matchingResult = customFibonacci is [1,
    ..];
```

The pattern shown in the example is a constant pattern, where direct numbers are employed in the sequence. Alternatively, relational patterns can be used, allowing the inclusion of comparison expressions, as demonstrated in the code below:

```
1. int[] customFibonacci = { 1, 2, 3, 5, 8 };
2. bool matchingResult = false;
```

```

3.
4. // Matching result is false, as customFibonacci ends with
   element 8
5. matchingResult = customFibonacci is [...,
   <8];
6.
7. // Matching result is true, 3 is the third element from the end
8. matchingResult = customFibonacci is [..., >=
   3, _, _];
9.
10. // Matching result is false, as the first element is not greater
   than 1
11.
12. matchingResult = customFibonacci is [>1,
   ..., 8];

```

The Var Pattern provides the flexibility to use the **var** keyword followed by a variable name, capturing the element present at that position in the sequence. This variable can then be utilized within the same scope for various purposes, offering versatility in pattern matching scenarios as we can see below:

```

1. int[] customFibonacci = { 1, 2, 3, 5, 8 };
2.
3. // Two elements are assigned to two variables
4. if (customFibonacci is [..., var
   penultimate, var last])
5. {
6.     Console.WriteLine($"penultimate =
   {penultimate}");
7.     Console.WriteLine($"last = {last}");
8. }
9. else
10. {
11.     Console.WriteLine("Pattern did not
   match!");
12. }
13.
14. //-----

```

```

15. // Case - Patterns are not matching
16. //-----
17. int[] someNumbers = { 1 };
18. if (someNumbers is [var firstValue, var
    secondValue, ..])
19. {
20.     Console.WriteLine($"firstValue =
        {firstValue}");
21.     Console.WriteLine($"secondValue =
        {secondValue}");
22. }
23. else
24. {
25.     Console.WriteLine("Pattern did not
        match!");
26. }

```

File-local types

The file modifier confines the scope and visibility of a top-level type to the file in which it is declared. Typically, the file modifier is applied to types generated by a source generator. File-local types offer a convenient solution for source generators to prevent name collisions among generated types. The file modifier designates a type as file-local, exemplified in the following instance:

```

1. file class ConfidentialComponent
2. {
3.     // implementation
4. }

```

Any types nested within a file-local type are likewise restricted to visibility within the file where it is declared. Other types in the assembly may share the same name as a file-local type without causing naming collisions.

A file-local type cannot serve as the return type or parameter type of any member with broader visibility than the file scope. Additionally, a file-local type cannot be a field member of a type with visibility exceeding file scope. Nevertheless, a more visible type may implicitly implement a file-local interface type. Explicit

implementation of a file-local interface is also possible, but such implementations can only be utilized within the file scope.

The subsequent illustration demonstrates a public type employing a confidential type within the file to offer a worker method.

Moreover, the public type implicitly implements a local interface defined within the file:

```
1. // In CustomFile.cs:
2. file interface ICustomFeature
3. {
4.     int OfferSolution();
5. }
6.
7. file class RestrictedComponent
8. {
9.     public int PerformTask() => 42;
10. }
11.
12. public class SpecialWidget : ICustomFeature
13. {
14.     public int OfferSolution()
15.     {
16.         var performer = new
            RestrictedComponent();
17.         return performer.PerformTask();
18.     }
19. }
```

In an alternate source file, it is possible to define types sharing identical names with the file-local types. However, the file-local types remain hidden and are not accessible, as we can see from the following code below:

```
1. // In AnotherFile.cs:
2. // No conflict with RestrictedComponent
3. // declared in CustomFile.cs
4. public class RestrictedComponent
5. {
6.     public void ExecuteTask()
```

```
7.     {  
8.         // omitted  
9.     }  
10. }
```

Required members

The concept of required members addresses certain well-known limitations associated with constructors. Constructors traditionally necessitate the caller to maintain the order or position of parameters, even if they do not require specific values within their implementation scope. This often results in the need for multiple constructors with different parameter combinations, leading to potential complexity. Additionally, any introduction of a new parameter affects all caller implementations. Required members offer a solution to these challenges by eliminating the necessity for multiple constructors and removing restrictions on parameter positions during object initialization. When utilizing required members, the compiler ensures that callers initialize these members within the object initialization scope.

However, there are limitations to the use of the required keyword. It cannot be applied to static properties or fields, as its design is specific to object initialization scope. Similarly, the required keyword is not applicable to private members since they are not visible to the caller. Additionally, it cannot be used for read-only members, as assignment is restricted to within the constructor.

With those changes we may have the following code:

```
1. public class CustomArticle  
2. {  
3.     public required string Headline { get;  
        set; }  
4.     public string? Subheading { get; set; }  
5.     public required string Writer { get;  
        set; }  
6.     public required DateTime ReleaseDate {  
        get; set; }  
7.  
8.     public override string ToString()
```

```

9.     {
10.         if
11.         (string.IsNullOrEmpty(Subheading))
12.         {
13.             return $"{Headline} by {Writer}
14.             ({ReleaseDate.ToShortDateString()})";
15.         }
16.         return $"{Headline}: {Subheading}
17.         by {Writer}
18.         ({ReleaseDate.ToShortDateString()})";
19.     }

```

And when creating a new instance of **CustomArticle** class, we must provide the required fields, as follows:

```

1. class Program
2. {
3.     static void Main()
4.     {
5.         // Instantiate CustomArticle
6.         var article = new CustomArticle
7.         {
8.             Headline = "Exploring C# 11
9.             Features",
10.            Subheading = "A Deep Dive into
11.            the Latest Language
12.            Enhancements",
13.            Writer = "John Doe",
14.            ReleaseDate = DateTime.Now
15.        };
16.
17.        // Display article details using ToString method
18.        Console.WriteLine(article.ToString());

```

```
17.     }  
18. }
```

In this example, a **CustomArticle** instance is created with specified values for **Headline**, **Subheading**, **Writer**, and **ReleaseDate**. The **ToString** method is then called to display the details of the article.

If we try to instantiate the **CustomArticle** without providing the required members, as the code below shows:

```
1. class Program  
2. {  
3.     static void Main()  
4.     {  
5.         // Instantiate CustomArticle  
6.         var article = new CustomArticle  
7.         {  
8.             Subheading = "A Deep Dive into  
the Latest Language  
9.                 Enhancements",  
10.            Writer = "John Doe",  
11.            ReleaseDate = DateTime.Now  
12.        };  
13.  
14.        // Display article details using ToString method  
15.        Console.WriteLine(article.ToString());  
16.    }  
17. }
```

We have the following error, as the figure shows:

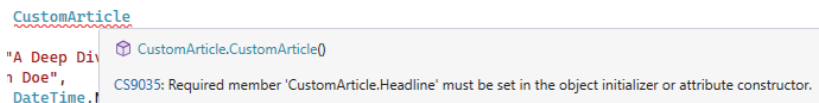


Figure 2.1: Warning when trying to initialize a class without required members

Auto-default structs

The C# 11 compiler guarantees the initialization of all fields in a

struct type to their default values during the execution of a constructor. This alteration ensures that any field or auto property not explicitly initialized by a constructor is automatically initialized by the compiler. Structs with constructors that do not definitively assign values to all fields can now be successfully compiled, and any fields not explicitly initialized will be set to their default values.

We can see an example of the changes in C# 11 in the example below:

```
1. public readonly struct ResultInfo
2. {
3.     public ResultInfo(double result)
4.     {
5.         ResultValue = result;
6.     }
7.
8.     public ResultInfo(double result, string
information)
9.     {
10.         ResultValue = result;
11.         Information = information;
12.     }
13.
14.     public ResultInfo(string information)
15.     {
16.         ResultValue = 0; // Default value for
double
17.         Information = information;
18.     }
19.
20.     public double ResultValue { get; init; }
21.     public string Information { get; init; } = "Default result";
22.
23.     public override string ToString() =>
    $"{ResultValue}
```

```

24.     ({Information})";
25. }
26.
27. public static void Main()
28. {
29.     var data1 = new ResultInfo(8.5);
30.     Console.WriteLine(data1);    // output: 8.5
                                   (Default result)
31.
32.     var data2 = new ResultInfo();
33.     Console.WriteLine(data2);    // output: 0
                                   (Default result)
34.
35.     var data3 = default(ResultInfo);
36.     Console.WriteLine(data3);    // output: 0
                                   (Default result)
37. }

```

Each struct inherently possesses a public parameterless constructor. If you choose to implement a parameterless constructor for a struct, it must be declared as public. In the case where a struct includes any field initializers, it is mandatory to explicitly declare a constructor, which may or may not be parameterless. If a struct declares a field initializer without any constructors, a compilation error is reported by the compiler. When a struct features an explicitly declared constructor, whether with parameters or parameterless, all field initializers for that struct are executed. Fields lacking a field initializer or an assignment in a constructor are automatically set to their default values.

Pattern match `Span<char>` on a constant string

For numerous releases, pattern matching has allowed testing whether a string holds a particular constant value. Presently, this pattern matching logic can be applied to variables of type `Span<char>` or `ReadOnlySpan<char>` as well.

Before the introduction of C# 11.0, it was essential to take a closer look at the reasons for using `Span<T>` and `ReadOnlySpan<T>`.

These types offer an efficient means of representing subsections of data. Specifically, if you were to slice out a portion of a string using `someString[4..10]`, the C# compiler would generate code that creates a new string object containing a copy of the requested characters. However, when employing a `ReadOnlySpan<char>`, no new copies of the data are generated. Instead, a new `ReadOnlySpan<char>` is created, pointing to the same underlying data with a reference to a slightly smaller portion.

Consider a scenario where the string to be examined is a substring of a larger string. In the absence of C# 11.0, utilizing examples like the ones mentioned would have required writing code that explicitly managed the substring, as we can see below:

```
1. string extractedName =  
    document[startIndex..endIndex];
```

However, with the introduction of C# 11.0, you have the option to express it differently:

```
1. ReadOnlySpan<char> extractedName =  
2.  
    doc.AsSpan[nameStartIndex..nameEndIndex];
```

The created span does not duplicate the data; instead, it points directly to the relevant portion within the doc string. This newfound capability in C# 11.0 allows you to leverage the `ReadOnlySpan<char>` named `extractedName` within patterns, as demonstrated in the previous examples.

`ReadOnlySpan<char>` offers a memory-efficient approach for handling substrings. It allows you to acquire a `ReadOnlySpan<char>` referring to a portion of a string or any char value sequence without the need for allocating a new object. With the enhancements introduced in C# 11.0, you can utilize the resulting `ReadOnlySpan<char>` seamlessly within a string constant pattern.

Extended nameof scope

Type parameter names and parameter names are now accessible within the scope of a `nameof` expression in an attribute declaration on a method. This enhancement allows the use of the `nameof` operator to indicate the name of a method parameter within an

attribute associated with either the method or the parameter declaration itself. This capability is particularly valuable for incorporating attributes related to nullable analysis.

Before the changes, we had to do something like this:

```
1. public class CustomClass
2. {
3.     [CustomDescription('customParam')]
4.     public void CustomMethod(string
        customParam)
5.     {
6.         var className = nameof(CustomClass);
7.         var methodName =
            nameof(CustomMethod);
8.         var paramName = nameof(customParam);
9.     }
10. }
```

Now, we can do this:

```
1. public class CustomClass
2. {
3.     [CustomDescription(nameof(customParam))]
4.     public void CustomMethod(string
        customParam)
5.     {
6.
7.         void
            LocalFunction<TGeneric>(TGeneric
                customParameter) { }
8.
9.         var customLambda =
            ([CustomDescription(nameof(parameter))]
10.             string
                parameter) =>
11.             Console.WriteLine(parameter);
12.     }
13. }
```

Numeric IntPtr and UIntPtr

The `nint` and `nuint` types are now synonymous with `System.IntPtr` and `System.UIntPtr`, respectively:

- `nint` is an alias for `IntPtr`, just as `int` is an alias for `Int32`.
- `nuint` is an alias for `UIntPtr`, similar to how `uint` is an alias for `UInt32`.

At the point of this writing, C# supports the following predetermined integer types:

C# type/keyword	Range	Size	.NET type
<code>sbyte</code>	-128 to 127	Signed 8-bit integer	System.SByte
<code>byte</code>	0 to 255	Unsigned 8-bit integer	System.Byte
<code>short</code>	-32,768 to 32,767	Signed 16-bit integer	System.Int16
<code>ushort</code>	0 to 65,535	Unsigned 16-bit integer	System.UInt16
<code>int</code>	-2,147,483,648 to 2,147,483,647	Signed 32-bit integer	System.Int32
<code>uint</code>	0 to 4,294,967,295	Unsigned 32-bit integer	System.UInt32
<code>long</code>	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed 64-bit integer	System.Int64
<code>ulong</code>	0 to 18,446,744,073,709,551,615	Unsigned 64-bit integer	System.UInt64
<code>nint</code>	Depends on platform (computed at runtime)	Signed 32-bit or 64-bit integer	System.IntPtr
<code>nuint</code>	Depends on platform (computed at runtime)	Unsigned 32-bit or 64-bit integer	System.UIntPtr

Figure 2.2: Integer types comparison

ref fields and scoped ref

Starting with the `ref` fields, this modification is a part of the *Low Level Struct Improvements*. While it may not find frequent use in the everyday tasks of the average programmer, its impact on performance within critical hot paths will be noticeable to all.

In earlier iterations of C#, the introduction of `Span<T>` empowered programmers to utilize direct memory access while benefiting from the safety of a garbage-collected language. However, this feature relied on an internal type called `ByReference<T>`, preventing programmers from implementing an equivalent of `Span` themselves.

With C# 11, the internal type **ByReference<T>** will undergo a transformation, becoming accessible to everyone through the definition of the type as a **ref struct** and **ref field**. Consequently, in C# 11, the internal Span will take on a form similar to the following as of today, as we can see in the code below:

```
1. readonly ref struct CustomSpan<T>
2. {
3.     readonly ref T _customField;
4.     readonly int _customLength;
5.
6.     public CustomSpan(ref T customValue)
7.     {
8.         _customField = ref customValue;
9.         _customLength = 1;
10.    }
11. }
```

About the **ref** scoped, the **ref** parameters have the option to incorporate the **scoped** modifier, signaling to the compiler that the reference is confined to the current method's lifetime. This limitation influences how the reference can be passed, stored, or assigned storage. These language improvements empower developers to craft more efficient code without resorting to unsafe features. With these enhancements, the compiler can enforce the lifetime rules of reference variables, as we can see in the example below:

```
1. CustomSpan<int> GenerateCustomSpan(scoped
   ref int customParameter)
2. {
3.     // method body
4. }
```

This enables us to pass the integer by reference, with the compiler ensuring that it is not stored anywhere within the function's scope. The objective is to prevent the creation of unnecessary pointers and handlers.

And we can see a bigger usage of both enhancements in the following code below, using both **ref** and **scoped ref**

functionalities:

```
1. ref struct CustomRefStruct
2. {
3.     public ref int InternalReference; // A
    ref field
4.
5.     public CustomRefStruct(ref int data)
6.     {
7.         InternalReference = ref data; //
        Assigning a ref field
8.     }
9. }
10.
11. public class UsageExample
12. {
13.     public void Demonstrate()
14.     {
15.         int dataValue = 99;
16.
17.         using CustomRefStruct customStruct
            =
18.             new CustomRefStruct(ref
                dataValue);
19.         scoped ref int localReference =
20.             ref
                customStruct.InternalReference; // Scoped ref
                variable
21.
22.         // Utilize localReference here
23.     }
24. }
```

Improved method group conversion to delegate

The C# standard regarding Method group conversions now encompasses the following aspects:

The conversion is allowed (though not mandatory) to utilize an

existing delegate instance that already encompasses these references.

In earlier versions of the standard, the compiler was restricted from reusing the delegate object generated for a method group conversion. With the C# 11 compiler, the delegate object produced from a method group conversion is cached and subsequently reused as a singular delegate object. This functionality was initially introduced as a preview feature in Visual Studio 2022 version 17.2 and in .NET 7 Preview 2.

An implicit conversion from a method group (§12.2) to a compatible delegate type (§20.4) is established when certain conditions are met. If *D* represents a delegate type and *E* is an expression classified as a method group, *D* is compatible with *E* only if *E* contains at least one method that can be invoked in its normal form (§12.6.4.2) for any argument list (§12.6.2). The argument list must have types and modifiers matching the parameter types and modifiers of *D*, as detailed below.

The compile-time application of the conversion from a method group *E* to a delegate type *D* involves the following steps:

1. Selection of a single method *M* corresponding to a method invocation (§12.8.9.2) of the form *E*(*A*), with modifications:
 - The argument list *A* consists of expressions, each classified as a variable and possessing the type and modifier (in, out, or ref) of the corresponding parameter in the **formal_parameter_list** of *D*. However, for parameters of type *dynamic*, the corresponding expression has the type *object* instead of *dynamic*.
 - Only candidate methods applicable in their normal form, without omitting any optional parameters (§12.6.4.2), are considered. Candidate methods are disregarded if applicable only in their expanded form or if one or more optional parameters lack a corresponding parameter in *D*.
4. Verification of the existence of a conversion by considering the algorithm of §12.8.9.2, which should produce a single best method *m* compatible (§20.4) with *D*.
5. If the selected method *m* is an instance method, the instance

expression associated with **E** identifies the target object of the delegate.

6. If the selected method **M** is an extension method denoted by a member access on an instance expression, that instance expression establishes the target object of the delegate.

The result of this conversion is a value of type **D**, signifying a delegate referring to the chosen method and its target object.

The following code example shows a method group conversion:

```
1. delegate string CustomDelegate1(object
   data);
2. delegate object CustomDelegate2(string
   input);
3. delegate object CustomDelegate3();
4. delegate string CustomDelegate4(object
   data, params object[] args);
5. delegate string CustomDelegate5(int value);
6.
7. class CustomTestClass
8. {
9.     static string CustomMethod(object
       input) {...}
10.
11.     static void CustomMethodGroup()
12.     {
13.         CustomDelegate1 delegate1 =
           CustomMethod;           // Ok
14.         CustomDelegate2 delegate2 =
           CustomMethod;           // Ok
15.         CustomDelegate3 delegate3 =
           CustomMethod;           // Error
16.                                     //-- not applicable
17.         CustomDelegate4 delegate4 =
           CustomMethod;           // Error
18.                                     //-- not applicable in normal form
19.         CustomDelegate5 delegate5 =
           CustomMethod;           // Error
```

```

20.                                     //– applicable but not compatible
21.     }
22. }

```

The assignment to `delegate1` involves an implicit conversion of the method group `CustomMethod` to a value of type `CustomDelegate1`.

The assignment to `delegate2` demonstrates the creation of a delegate for a method with less derived (contravariant) parameter types and a more derived (covariant) return type.

In contrast, the assignment to `delegate3` highlights the absence of conversion when the method is not applicable.

For `delegate4`, the assignment illustrates the requirement that the method must be applicable in its normal form.

Finally, the assignment to `delegate5` exemplifies how parameter and return types of the delegate and method are permitted to differ only for reference types.

Warning wave 7

In each release of the C# compiler, new warnings and errors may be introduced. When these new warnings could potentially be reported on existing code, they are introduced under an opt-in system known as a **warning wave**. This opt-in system ensures that new warnings will not be displayed on existing code unless you take explicit action to enable them. The activation of warning waves is accomplished using the `AnalysisLevel` element in your project file. If `<TreatWarningsAsErrors>true</TreatWarningsAsErrors>` is specified, warnings from enabled warning waves will be treated as errors. Warning wave 5 diagnostics were incorporated in C# 9, wave 6 diagnostics in C# 10, and wave 7 diagnostics in C# 11.

In C# 11 we had the following addition:

- CS8981: The type name only contains lower-cased ascii characters.
- Any additional keywords introduced in C# will consist solely of lowercase ASCII characters. ASCII abbreviated from American Standard Code for Information Interchange, is a character encoding standard for electronic

communication.

This warning serves to prevent conflicts between your types and any potential future keywords. To resolve this warning, consider renaming the type to incorporate at least one non-lowercase ASCII character. This could involve using an uppercase character, a digit, or an underscore. The provided code results in CS8981:

```
1. public class samplelowercaseclassname  
2. {  
3. }
```

C# 12 updates

In this section, we delve into the notable updates introduced in C# 12, aimed at enhancing developer productivity and code expressiveness. From the introduction of pattern-based interfaces to the integration of global using directives, C# 12 introduces several features designed to streamline development workflows and improve code readability. Developers will explore pattern-based interfaces, allowing for more flexible type declarations and improved code organization. Additionally, the integration of global using directives simplifies namespace management, reducing boilerplate code and enhancing code clarity. Understanding these changes empowers developers to leverage the latest language features effectively, enabling them to write cleaner, more concise, and maintainable code in their C# projects.

Primary constructors

Now, primary constructors are no longer exclusive to record types; they can be created in any class or struct. These constructors allow parameters to be accessible throughout the entire class body. Explicitly declared constructors must call the primary constructor using the `this()` syntax to ensure that all parameters are definitively assigned. Introducing a primary constructor in a class prevents the compiler from generating an implicit parameterless constructor. In structs, the implicit parameterless constructor initializes all fields, including primary constructor parameters, to the 0-bit pattern. However, the compiler only generates public properties for primary constructor parameters in record types, not in non-record classes or structs.

We can see an example of primary constructor in the code block below:

```
1. public class Client (string clientName)
2. {
3.     public string Name { get; } = clientName;
   // compiler warning
4.
5.     public string FullName => clientName;
6.
7.     public void ModifyName() => clientName =
   clientName.ToUpper();
8. }
9.
10. // Console app
11. var client = new Client("Thiago");
12. client.ModifyName();
13. Console.WriteLine($"Name: {client.Name}");
   // Thiago
14. Console.WriteLine($"Full Name:
   {client.FullName}"); // THIAGO
```

Collection expressions

A collection expression allows for concise creation of common collection values. It's a succinct syntax enclosed in [and] brackets, suitable for assigning to various collection types. For instance, consider initializing a **System.Span<T>** of string elements representing the months of the year:

```
1. Span<string> monthsOfYear = ["Jan", "Feb",
   "Mar", "Apr", "May",
2.     "Jun", "Jul", "Aug", "Sep",
   "Oct", "Nov", "Dec"];
3. foreach (var month in monthsOfYear)
4. {
5.     Console.WriteLine(month);
6. }
```

The spread operator, denoted by .. in a collection expression,

replaces its argument with the elements from that collection. This argument must be of a collection type. Below are examples illustrating the functionality of the spread operator:

```
1. int[] evenNumbers = [2, 4, 6, 8];
2. int[] oddNumbers = [1, 3, 5, 7, 9];
3. int[] allNumbers = [..evenNumbers,
   ..oddNumbers];
4.
5. foreach (var number in allNumbers)
6. {
7.     Console.Write($"{number}, ");
8. }
9. // Output:
10. // 2, 4, 6, 8, 1, 3, 5, 7, 9,
```

Inline arrays

Inline arrays enhance application performance by enabling the creation of fixed-size arrays within struct types. Typically utilized by runtime and library developers, they offer performance similar to unsafe fixed-size buffers. While developers may not directly declare inline arrays, they benefit from them when accessing

System.Span<T> or **System.ReadOnlySpan<T>** objects via runtime APIs. Below is an example declaration of an inline array within a struct:

```
1. [System.Runtime.CompilerServices.InlineArray(7)]
2. public struct SampleArray
3. {
4.     private int _element0;
5. }
```

Usage is identical to standard arrays:

```
1. var sampleArray = new SampleArray ();
2. for (int i = 0; i < 7; i++)
3. {
4.     sampleArray [i] = i;
5. }
6.
```

```
7. foreach (var i in sampleArray )
8. {
9.     Console.WriteLine(i);
10. }
```

Default lambda parameters

Lambda expressions now support default parameter values, adhering to the same syntax and conventions as regular method or local function arguments, as we can see from the example below:

```
1. // Define a lambda expression to increment a number by a
   // specified amount (with a default increment of 1)
2. var IncrementValue = (int number, int amount = 1)
   => number
3.
   + amount;
4.
5. // Call the lambda expression with different numbers and see
6. // the results
7. Console.WriteLine(IncrementValue(10));
   // Output: 11
8. Console.WriteLine(IncrementValue(10, 5));
   // Output: 15
```

You can also define lambda expressions with parameter arrays using the **params** keyword:

```
1. var calculateSum = (params int[] numbers) =>
2. {
3.     int totalSum = 0;
4.     foreach (var num in numbers)
5.         totalSum += num;
6.
7.     return totalSum;
8. };
9.
10. var emptySum = calculateSum();
11. Console.WriteLine(emptySum); // 0
12.
```

```
13. var sequenceNumbers = new[] { 1, 2, 3, 4, 5 };
14. var totalSum = calculateSum(sequenceNumbers);
15. Console.WriteLine(totalSum); // 15
```

With these updates, when a method group containing a default parameter is assigned to a lambda expression, the lambda expression also inherits the same default parameter. Likewise, a method group with a params array parameter can be assigned to a lambda expression.

Lambda expressions featuring default parameters or params arrays don't directly correspond to `Func<>` or `Action<>` types. However, you can define delegate types that include default parameter values:

```
1. delegate int AdditionDelegate(int value1, int
   value2 = 1);
2. delegate int TotalDelegate(params int[]
   numbers);
```

Alternatively, you can use implicitly typed variables with `var` declarations to define the delegate type. In this case, the compiler determines the appropriate delegate type.

ref readonly parameters

The `in` modifier enables the compiler to generate a temporary variable for the argument and pass a readonly reference to that variable. This temporary variable creation occurs when the argument requires conversion, implicates an implicit conversion from the argument type, or represents a value that is not a variable (such as a literal value or property accessor return). In cases where the API mandates passing arguments by reference, it's advisable to utilize the `ref readonly` modifier instead of `in`.

Methods utilizing `in` parameters potentially benefit from performance optimizations, especially with large struct type arguments. By indicating that arguments can be passed by reference safely, methods declared with `in` parameters avoid costly copies in tight loops or critical code paths where performance is crucial. Explicitly specifying `in` at the call site ensures the argument is passed by reference, not by value, offering two key benefits:

- Forcing the compiler to select a method defined with a

matching in parameter when multiple methods differ only in the presence of `in`, favoring the by reference overload.

- Declaring the intent to pass an argument by reference, indicating that the argument used with `in` must represent a directly referable location.

Omitting `in` at the call site prompts the compiler to create a temporary variable for passing by read-only reference to the method, overcoming restrictions associated with `in` arguments such as compile-time constants, properties, or expressions requiring implicit conversion from the argument type to the parameter type. In such cases, the compiler generates a temporary variable to store the value of the constant, property, or expression.

The `ref readonly` modifier is mandatory in the method declaration, while it is optional at the call site. Either the `in` or `ref` modifier can be employed. However, the `ref readonly` modifier isn't permissible at the call site. The choice of modifier at the call site can elucidate the characteristics of the argument.

- Use `ref` only if the argument is a writable variable.
- Use `in` when the argument is a variable, regardless of whether it is writable or read-only.
- Neither modifier can be added if the argument is not a variable but an expression.

The subsequent examples illustrate these conditions. Consider the following method utilizing the `ref readonly` modifier to indicate that a large struct should be passed by reference for performance reasons:

```
1. public static void UpdateStruct (ref readonly  
   OptionStruct target)  
2. {  
3. // Implementation details omitted  
4. }
```

You can invoke the method with either the `ref` or `in` modifier. Omitting the modifier triggers a compiler warning. However, if the argument is an expression rather than a variable, you cannot apply the `in` or `ref` modifiers. In such cases, it's advisable to suppress the warning.

```
1. UpdateStruct (in struct);
```


2. `UpdateStruct (ref struct) ;`
3. `UpdateStruct (struct) ;`
4. *// Warning! variable should be passed with `ref` or `in`*
5. `UpdateStruct (new OptionStruct());`
6. *// Warning, but an expression, so no variable to reference*

If the variable is declared as readonly, you must use the `in` modifier; using the `ref` modifier in this context will result in a compiler error.

The `ref readonly` modifier denotes that the method anticipates the argument to be a variable, not an expression that is not a variable. Expressions that are not variables include constants, method return values, and properties. When the argument is not a variable, the compiler emits a warning.

Alias any type

In C# 12, the `using` alias directive has been enhanced to permit aliasing any type, not limited to named types. This flexibility allows aliasing tuples, pointers, array types, generic types, and more. Consequently, you can now employ a concise, descriptive alias instead of the full structural representation of a tuple throughout your codebase.

Consider the following example of aliasing a tuple. Initially, declare the alias:

1. `using SampleTuple = (int x, int y);`

Subsequently, utilize it as you would any other type. Employ it as a return type, within the parameters list of a method, or even for instantiating new instances of that type. The possibilities are virtually limitless.

Here is an example demonstrating the usage of the tuple alias declared earlier:

1. `SampleTuple CopyTuple(SampleTuple input)`
2. `{`
3. `return new SampleTuple(input.x, input.y);`
4. `}`

Conclusion

In conclusion, this chapter has provided an in-depth exploration of the main changes introduced in C# 11 and C# 12. From primary constructors to inline arrays, we have examined a wide range of enhancements designed to improve developer productivity and code quality.

By understanding these new features and their applications, developers can unlock the full potential of C# and stay ahead in the rapidly evolving landscape of software development. Whether it is leveraging collection expressions for concise code or embracing optional parameters in lambda expressions for greater flexibility, the advancements in C# 11 and C# 12 offer exciting opportunities for innovation and optimization.

As we embrace these changes and continue to evolve with the language, we look forward to seeing the creative solutions and groundbreaking applications that developers will build using these new capabilities.

In the next chapter *Mastering Entity Framework Core*, we will discuss the intricacies of data access and manipulation. In this exploration, we will unravel the capabilities of Entity Framework Core, a powerful and **versatile Object-Relational Mapping (ORM)** framework. From understanding its core concepts to mastering advanced features, this chapter promises to equip you with the knowledge and skills needed to leverage Entity Framework Core for seamless interaction with databases. Join us as we navigate through essential concepts, best practices, and hands-on applications, empowering you to become a proficient master of Entity Framework Core.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 3

Mastering Entity Framework Core

Introduction

This chapter provides a comprehensive guide to using this powerful **object-relational mapping (ORM)** tool for .NET development. The chapter covers the two primary approaches for building a data model in Entity Framework Core: **Database-First** and **Code-First**.

The chapter delves into Database-First modeling, which involves creating a database schema before generating a data model. It provides step-by-step instructions for using Entity Framework Core tools to reverse-engineer a database schema and create a corresponding data model. The chapter also covers Code-First modeling, which allows developers to design a data model and have Entity Framework Core generate a database schema.

The chapter also provides an overview of data modeling concepts and explains how to define data models using Entity Framework Core. It covers the use of data annotations to define data models and explains how to use them to customize data model generation.

Additionally, the chapter covers data management using Entity

Framework Core, including querying and modifying data in a database. It explains how to use **Language Integrated Query (LINQ)** to Entities to write queries and provides examples of common query operations. The chapter also covers data modification operations, such as inserting, updating, and deleting data.

Structure

This chapter covers the following topics:

- Mastering Entity Framework Core
- Database First
- Code First
- LINQ to Entities
- Data Annotations
- Data Modeling
- Data Management

Objectives

Upon completing this chapter, you will gain a comprehensive understanding of **Entity Framework Core (EF Core)**, a vital tool for database management in your applications. Delve into Database First and Code First paradigms, honing the ability to generate entities from schema and orchestrate database migrations. Explore nuanced data modeling, leveraging LINQ to Entities for intricate queries and performance optimization. Grasp the significance of Data Annotations in mapping entities and enforcing validation rules. Navigate complex relationship structures, ensuring transactional integrity and concurrency control. Equipped with these skills, you'll adeptly navigate both Database First and Code First workflows, adeptly managing data operations in your applications with EF Core.

Mastering Entity Framework Core

Entity Framework Core is a lightweight, cross-platform ORM framework for .NET applications. It simplifies the process of

accessing and manipulating data from relational databases by providing a high-level abstraction layer. With Entity Framework Core, developers can define a conceptual model using classes and relationships, and the framework handles the underlying database operations. It supports various database providers, allowing developers to work with different database systems seamlessly. Entity Framework Core offers features like query optimization, change tracking, and migration support, making it a powerful tool for efficient data access and management in modern application development.

Database First

The **Database First** approach in Entity Framework Core is utilized when we need to reverse engineer our existing database model into classes that can be consumed by our application. It offers a valuable capability to seamlessly connect new projects with pre-existing databases, making it particularly useful during technology migrations.

Database First enables the translation of database objects into corresponding classes within your project. This approach is supported by all Entity Framework Core providers and effectively reflects data fields, data types, and table relationships from various data sources into your project's language.

When integrating an existing database with a new project, the Database First approach is highly recommended as it saves time by automatically generating the necessary classes for you.

Benefits of using Database First

One of the primary advantages of the Database First approach is its ability to seamlessly connect new projects with existing databases, regardless of the technologies previously used. Additionally, it allows for the parallel development of the project's architecture and database modeling, enabling different teams to work concurrently. Subsequently, the project can be connected to the created database using the Database First approach.

If, after successfully creating the entities in your project using the Database First approach, you need to update any object within your

project or database, you have the following options:

- If you make changes to your code and want them to reflect in the database, you should use the Migration Commands feature.
- If you make changes to your database and want them to be reflected in your code, you need to scaffold the database again.

Implementation step-by-step

In this implementation, we are using an existing and public database, the Northwind Database. You can create a copy of the Northwind Database, in this example, we will be naming the database as **SampleThiagoDb**, by executing the following script SQL and steps:

<https://raw.githubusercontent.com/microsoft/sql-server-samples/master/samples/databases/northwind-pubs/instnwnd.sql>

1. First, create a console application targeting .NET 8.0 and add the following Nuget Packages needed to scaffold and connect to the database:
 - Microsoft.EntityFrameworkCore.Design
 - Microsoft.EntityFrameworkCore.Tools
 - Microsoft.EntityFrameworkCore.SqlServer
5. Get your connection string to the Northwind Database. You can easily retrieve your connection string from the SQL Server Object Explorer Window then go to the properties of your Database, as you can see from the picture below:

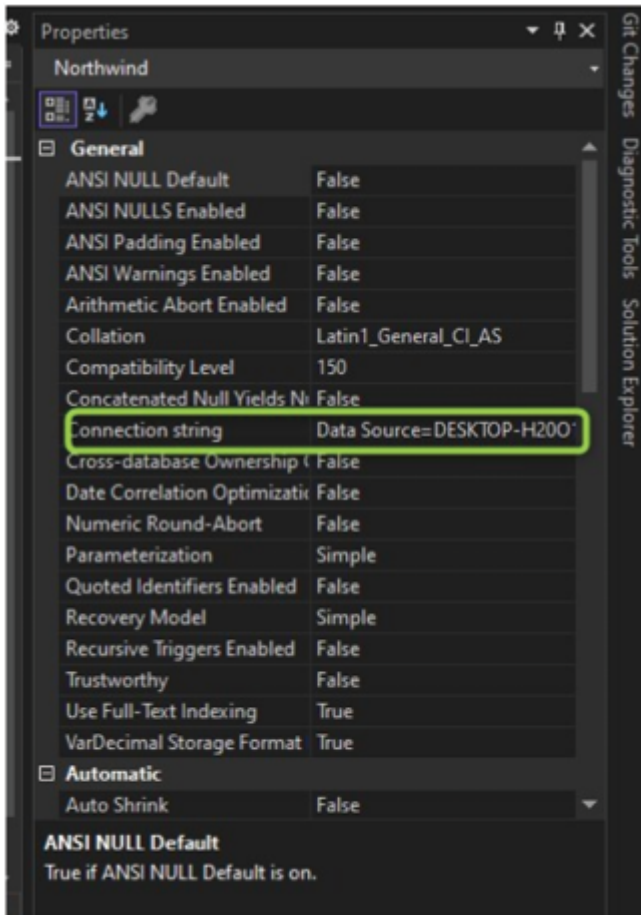


Figure 3.1: How to find your connection string

1. Open the Package Manager Console by opening the View menu, Other Windows, and clicking on Package Manager Console and execute the following command:

```
1. Scaffold-DbContext {Your Connection String}  
Microsoft.EntityFrameworkCore.SqlServer
```

The output is the one as follows, with the **DbContext** created and classes scaffolded:

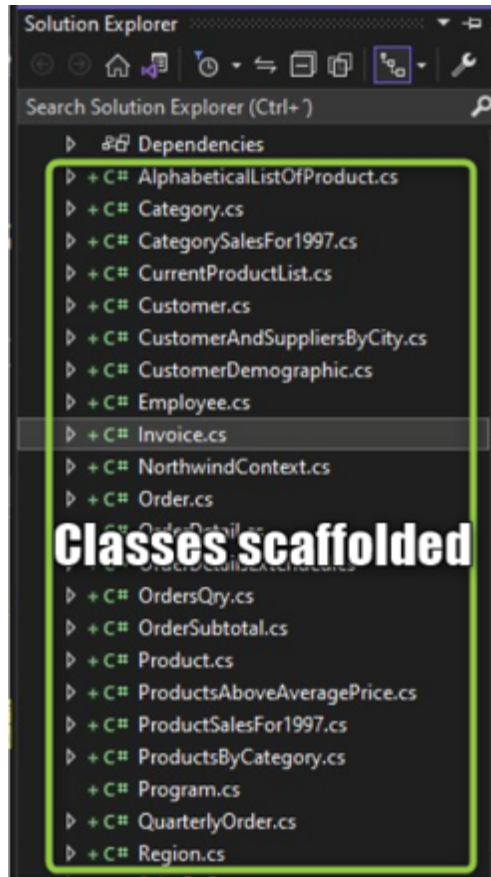


Figure 3.2: Classes scaffolded after executing the Scaffold command

Code First

Entity Framework Core Code First simplifies database development by allowing you to define your data model using C# classes and then automatically generating the corresponding database schema. This approach streamlines the process of creating and maintaining databases for your applications.

With Entity Framework Core's Code First approach, you can focus on designing your application's entities and relationships using familiar object-oriented principles. By leveraging attributes or fluent API configuration, you can fine-tune various aspects of your data model, such as specifying primary keys, defining relationships, and

applying constraints.

Benefits of using Code First

The benefits of Code First extend beyond initial database creation. It offers seamless database migration support, enabling you to evolve your data model over time without losing existing data. You can easily add new entities, modify existing ones, and update the database schema to reflect these changes using migrations.

The code-first approach provides a flexible and efficient way to manage your database schema alongside your application's development process. By leveraging the simplicity of C# classes and automated schema generation, you can focus on building robust applications while letting Entity Framework Core handle the complexities of the underlying database.

Implementation step-by-step

In this implementation we are setting up a database with a few entities using Entity Framework Core with the Code First.

First, create a Web Application targeting .NET 8.0 and add the following Nuget Packages needed to connect and manage the database:

- Microsoft.EntityFrameworkCore.Tools
- Microsoft.EntityFrameworkCore
- Microsoft.EntityFrameworkCore.SqlServer

Create a few classes, as follows:

```
1. public class Supermarket
2. {
3.     public int Id { get; set; }
4.     public string Address { get; set; }
5.     public string City { get; set; }
6.     public ICollection<Product>
       Products { get; set; }
7. }
8. public class Product
9. {
```

```

10.         public int Id { get; set; }
11.         public string Name { get; set; }
12.         public double Price { get; set; }
13.         public string Description { get;
set; }
14.     }
15.     public class Customer
16.     {
17.         public int Id { get; set; }
18.         public string Name { get; set; }
19.         public string Address { get; set; }
20.         public int Age { get; set; }
21.         public ICollection<Product>
Products { get; set; }
22.     }

```

1. Create the **DbContext** as **SampleDbContext**:

```

1.     public class SampleDbContext : DbContext
2.     {
3.         public
SampleDbContext(DbContextOptions<SampleDbContext>
options)
4.             : base(options)
5.         {
6.         }
7.
8.         public DbSet<Product> Products {
get; set; }
9.         public DbSet<Customer> Customers {
get; set; }
10.        public DbSet<Supermarket>
Supermarkets { get; set; }
11.    }

```

2. Update the **Program.cs** to configure dependency injection:

```

1. builder.Services.AddDbContext<SampleDbContext>(
2.     options =>
options.UseSqlServer("Data Source=DESKTOP-

```

```
H20012E;Initial
Catalog=SampleThiagoDb;Integrated
Security=True;Connect
Timeout=30;Encrypt=False;Trust Server
Certificate=False;Application
Intent=ReadWrite;Multi Subnet
Failover=False"));
```

3. Open the Package Console Manager and run the following command:

1. Add-Migration InitialCreate

As output, you should have the migrations folder created with the following files:

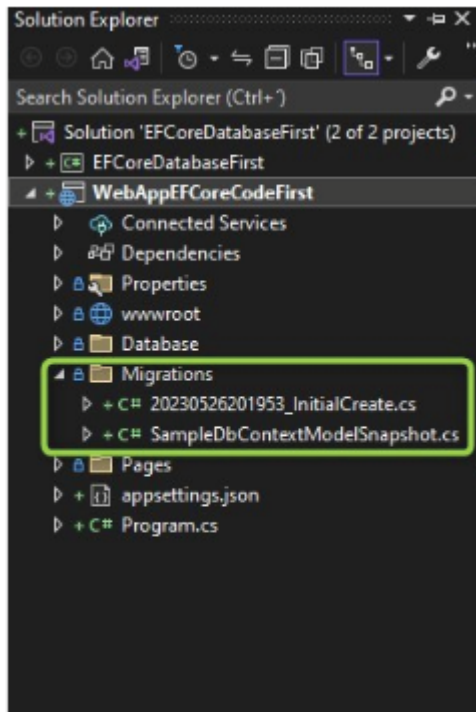


Figure 3.3: Migrations folder and class created

1. Now, we must execute the following command in the Package Manager Console, which will be responsible for creating the database:

1. update-database

After successfully completing you must have your database created:

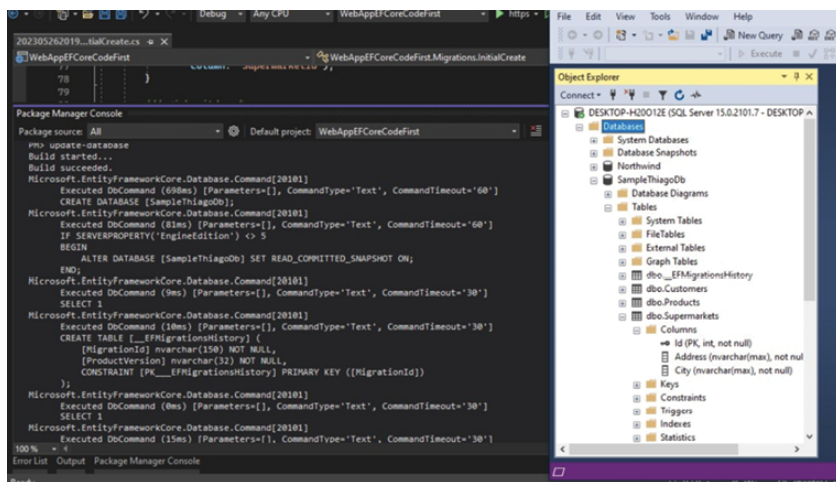


Figure 3.4: Database created after executing the update-database command

LINQ to Entities

LINQ to Entities allows developers to write queries using familiar programming languages such as C# or Visual Basic, making it easier to express complex data retrieval and manipulation operations. With LINQ to Entities, developers can leverage the full power of LINQ to compose expressive and type-safe queries. LINQ to Entities supports a wide range of operations, including filtering, sorting, grouping, and aggregating data. It also supports complex join operations and supports various LINQ operators and methods for data manipulation and transformation.

When using LINQ to Entities, queries are represented by **IQueryable<T>** objects. These objects define a query that can be executed against the database. The LINQ to Entities provider translates the LINQ expressions and operations into SQL queries that are executed on the underlying database.

One of the key advantages of LINQ to Entities is its integration with Entity Framework Core and its components, such as change tracking, transaction management, and database updates. This integration allows developers to write efficient and optimized queries that take advantage of the underlying capabilities of Entity

Framework Core.

Here is the step-by-step procedure running on the background every time that we execute a LINQ to Entities query:

1. Creates a new instance of the `ObjectQuery<T>` object from the `ObjectContext`

The `ObjectQuery` object represents a typed query where the `T` is the type of the entity that will be returned in the result.

2. Compose a LINQ To Entities query based on the `ObjectQuery<T>` instance.

The `ObjectQuery<T>` class, implementing `IQueryable<T>`, acts as the data source for LINQ to Entities queries, specifying the desired data to retrieve and allowing sorting, grouping, and shaping. Queries are stored in variables without taking immediate action or returning data until executed. LINQ to Entities supports two syntaxes: query expression and method-based, and LinQ was introduced back in C# 3.0 and Visual Basic 9.0.

3. Convert the LINQ code into command trees.

First the LINQ code must first be converted to a command tree representation specific to the framework. LINQ queries consist of standard operators and expressions, where operators are methods on a class and expressions can contain a broad range of functionality. The Entity Framework unifies both operators and expressions into a single type of hierarchy within a command tree, which is then used for query execution. The conversion process involves transforming both the query operators and expressions accordingly.

4. Execute the query, represented by command trees, against the data source.

During query execution, all query expressions or components are evaluated either on the client or the server, depending on the context.

5. The query materialization returning the result.

The query results are delivered to the client as **Common Language Runtime (CLR)** types, never returned as raw data records. The CLR type can be defined by the user, the Entity Framework, or generated as anonymous types by the

compiler. The Entity Framework handles all object materialization operations.

Data Annotations

Data Annotations are attributes applied to Classes or Classes' attributes that serves to modify or override the configuration derived from model building conventions, while the configuration achieved through mapping attributes can be further overridden using the model building API in **OnModelCreating**. To ensure clarity and simplicity in their use, mapping attributes designed for EF Core are exclusive to EF Core, avoiding any semantic variations across technologies.

Data annotations has a higher priority over conventions, but the Fluent API configuration has more priority than Data Annotations. So, your Data Annotations can override the conventions configurations, but the Fluent API Configuration can override the data annotations.

Most common Data Annotations

Below you can find the list of the most used Data Annotations:

- **Table**: Sets the table name for an Entity.
- **Column**: Sets the column name, type, and the column order.
- **MaxLength**: Sets the property max length.
- **Required**: Sets the property as mandatory.
- **ForeignKey**: Sets the relationship between entities.
- **NotMapped**: Exclude the property to the database model.

Applying Data Annotations

We will be applying the Data Annotations above to our project created in the previous Code-First Implementation:

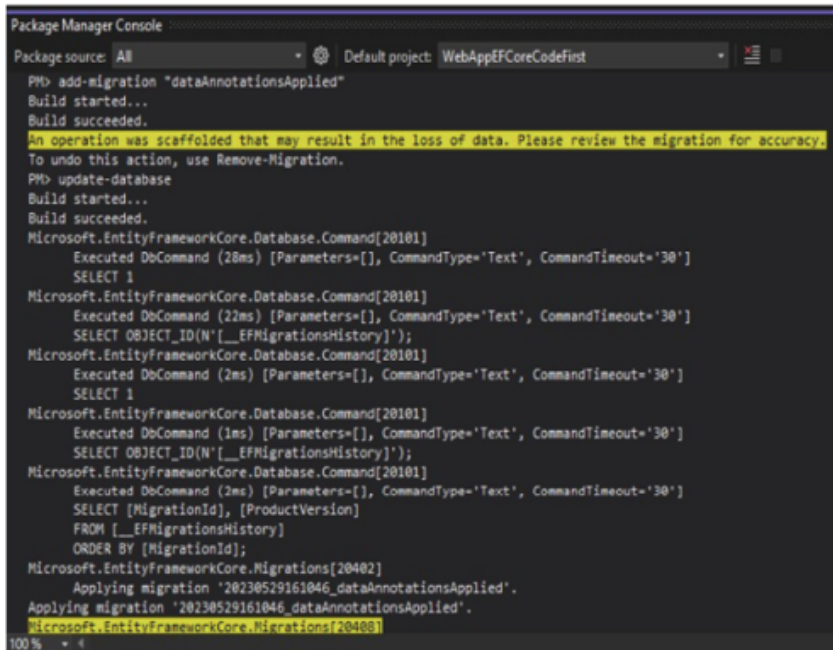
```
1. public class Supermarket
2. {
3.     public int Id { get; set; }
4.     public string Address { get; set; }
5.     [Required(ErrorMessage = "The City
```

```

is required.")]
6.         public string City { get; set; }
7.         public ICollection<Product>
Products { get; set; }
8.     }
9.     public class Product
10.    {
11.        public int Id { get; set; }
12.        public string Name { get; set; }
13.        [Column(TypeName = "money")]
14.        public double Price { get; set; }
15.        [MaxLength(50, ErrorMessage = "Max
length is 50!")]
16.        public string Description { get;
set; }
17.    }
18.    public class Customer
19.    {
20.        public int Id { get; set; }
21.        [Column("CustomerName", Order = 5)]
22.        public string Name { get; set; }
23.        public string Address { get; set; }
24.        [Precision(2)]
25.        public int Age { get; set; }
26.        public ICollection<Product>
Products { get; set; }
27.    }

```

To see those changes reflecting in the database we must run the “**add-migration**” command followed by the “**update-database**” command:



```
Package Manager Console
Package source: All Default project: WebAppEFCoreCodefirst
PM> add-migration "dataAnnotationsApplied"
Build started...
Build succeeded.
An operation was scaffolded that may result in the loss of data. Please review the migration for accuracy.
To undo this action, use Remove-Migration.
PM> update-database
Build started...
Build succeeded.
Microsoft.EntityFrameworkCore.Database.Command[20101]
    Executed DbCommand (28ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
    SELECT 1
Microsoft.EntityFrameworkCore.Database.Command[20101]
    Executed DbCommand (22ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
    SELECT OBJECT_ID(N'[__EFMigrationsHistory]');
Microsoft.EntityFrameworkCore.Database.Command[20101]
    Executed DbCommand (2ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
    SELECT 1
Microsoft.EntityFrameworkCore.Database.Command[20101]
    Executed DbCommand (1ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
    SELECT OBJECT_ID(N'[__EFMigrationsHistory]');
Microsoft.EntityFrameworkCore.Database.Command[20101]
    Executed DbCommand (2ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
    SELECT [MigrationId], [ProductVersion]
    FROM [__EFMigrationsHistory]
    ORDER BY [MigrationId];
Microsoft.EntityFrameworkCore.Migrations[20402]
    Applying migration '20230529161046_dataAnnotationsApplied'.
Applying migration '20230529161046_dataAnnotationsApplied'.
Microsoft.EntityFrameworkCore.Migrations[20403]
```

Figure 3.5: Package Manager Console with the output from the EF Core commands

In the following figure, database is reflecting the changes made in the code:

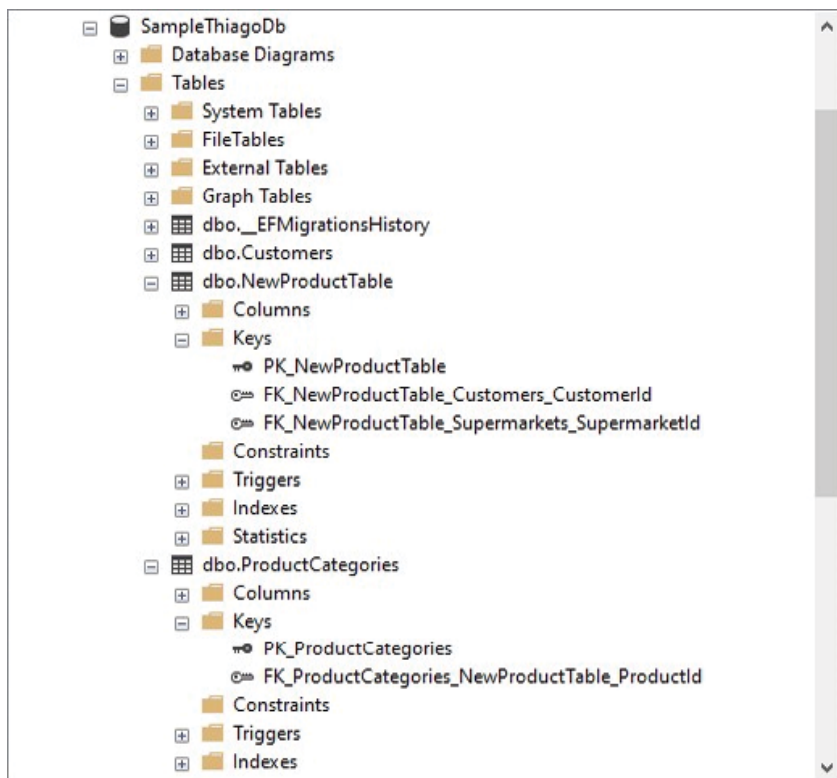


Figure 3.6: Database reflecting the changes made in the code

Data Modeling

Data Modeling is a fundamental aspect that focuses on defining relationships between entities. These relationships are crucial for organizing and representing data effectively. Three common types of relationships are explored in this sub-topic: one-to-one, one-to-many, and many-to-many.

Understanding these relationships and their implementation in Entity Framework Core is essential for designing a robust and efficient data model. By establishing the appropriate relationships, developers can create a well-structured and interconnected entity framework that accurately represents the real-world data requirements of their applications.

The relationships are described below, with a practical example of them working in our previous project created for the Code-First

implementation and updated with the Data Annotations examples.

For a better demonstration of the following relationships, the class **ProductCategory** has been created:

```
1. public class ProductCategory
2. {
3.     public int Id { get; set; }
4.     public string Name { get; set; }
5.     public string Description { get;
    set; }
6. }
```

One-to-one relationship

A **one-to-one relationship** indicates that one instance of an entity is directly associated with another instance. This type of relationship establishes a direct connection between the entities, enabling them to share information seamlessly. We can understand better about the one-to-one relationship types below:

Required one-to-one

In this scenario, it is required that a **Product** is associated to a **ProductCategory**:

- `[Table("NewProductTable")]`
- `public class Product`
- `{`
- `public int Id { get; set; }`
- `public string Name { get; set; }`
- `[Column(TableName = "money")]`
- `public double Price { get; set; }`
- `[MaxLength(50, ErrorMessage = "Max length is 50!")]`
- `public string Description { get; set; }`
- `public ProductCategory? Category`

- { get; set; }
- }
- public class ProductCategory
- {
- public int Id { get; set; }
- public string Name { get; set; }
- public string Description { get; set; }
- public int ProductId { get; set; }
- public Product Product { get; set; } = null!;
- }

After running the “**add-migration**” command followed by the “**update-database**” command, we have those changes in the database:

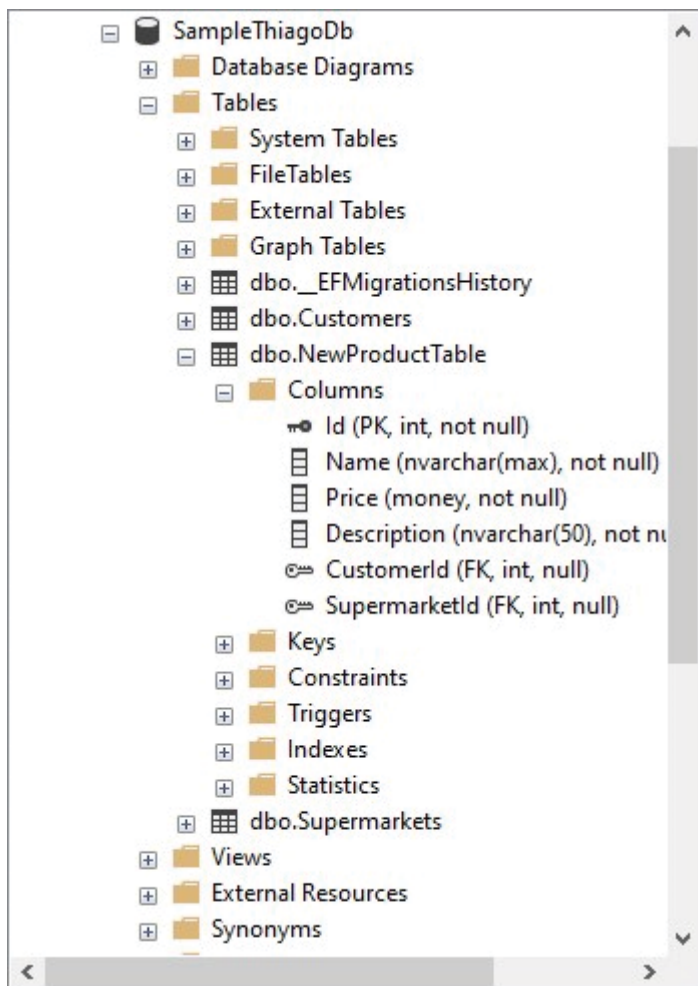


Figure 3.7: Database reflecting the changes made in the code.

In a required one-to-one relationship, the principal entity should have one or more primary or alternate key properties, such as **Product.Id**. The dependent entity, on the other hand, should have one or more foreign key properties, such as **ProductCategory.ProductId**. Optionally, the principal entity can have a reference navigation pointing to the dependent entity, like **Product.Category**. Similarly, the dependent entity can also have an optional reference navigation pointing back to the principal entity, such as **ProductCategory.Product**. These components together define and establish a one-to-one relationship between

entities in Entity Framework Core.

Optional one-to-one

In this scenario, we may have a Product that is associated to a **ProductCategory**, and we also may have Product that is not associated to any **ProductCategory**, as we can see from the code below:

```
1. [Table("NewProductTable")]
2. public class Product
3. {
4.     public int Id { get; set; }
5.     public string Name { get; set; }
6.     [Column(TypeName = "money")]
7.     public double Price { get; set; }
8.     [MaxLength(50, ErrorMessage = "Max
length is 50!")]
9.     public string Description { get; set; }
10.    public ProductCategory? Category { get;
set; }
11. }
12. public class ProductCategory
13. {
14.     public int Id { get; set; }
15.     public string Name { get; set; }
16.     public string Description { get; set; }
17.     public int? ProductId { get; set; }
18.     public Product? Product { get; set; }
19. }
```

After running the “**add-migration**” command followed by the “**update-database**” command, we have those changes in the database:

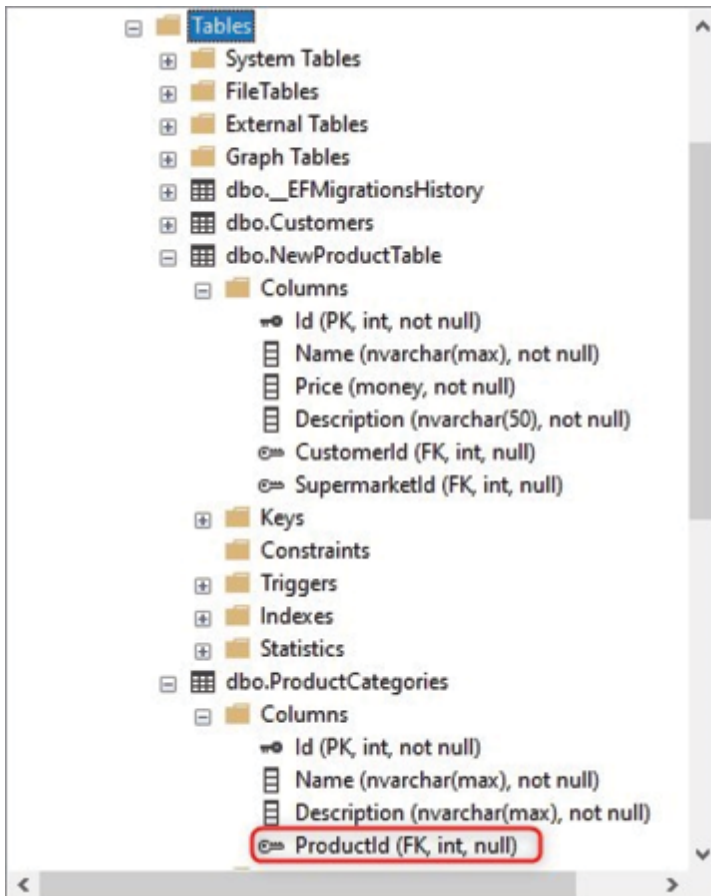


Figure 3.8: The column *ProductId* nullable reflecting the changes in the code

In an optional one-to-one relationship, we have nullable navigation properties between the entities. We have the **ProductCategory.Product**, **ProductCategory.ProductId**, and **Product.ProductCategory** nullable

One-to-many relationship

In a **one-to-many relationship**, a single instance of an entity is related to multiple instances of another entity. This relationship enables efficient data organization, as the "one" entity can maintain a collection of related instances from the "many" entity.

Required one-to-many

In this scenario, a **Product** is associated from one until N **ProductCategory**:

```
1. [Table("NewProductTable")]
2. public class Product
3. {
4.     public int Id { get; set; }
5.     public string Name { get; set; }
6.     [Column(TableName = "money")]
7.     public double Price { get; set; }
8.     [MaxLength(50, ErrorMessage = "Max
length is 50!")]
9.     public string Description { get;
set; }
10.     public ICollection<ProductCategory>
Category { get; } = new
List<ProductCategory>();
11. }
12. public class ProductCategory
13. {
14.     public int Id { get; set; }
15.     public string Name { get; set; }
16.     public string Description { get;
set; }
17.     public int ProductId { get; set; }
18.     public Product Product { get; set;
} = null!;
19. }
```

After running the “**add-migration**” command followed by the “**update-database**” command, we have those changes in the database:

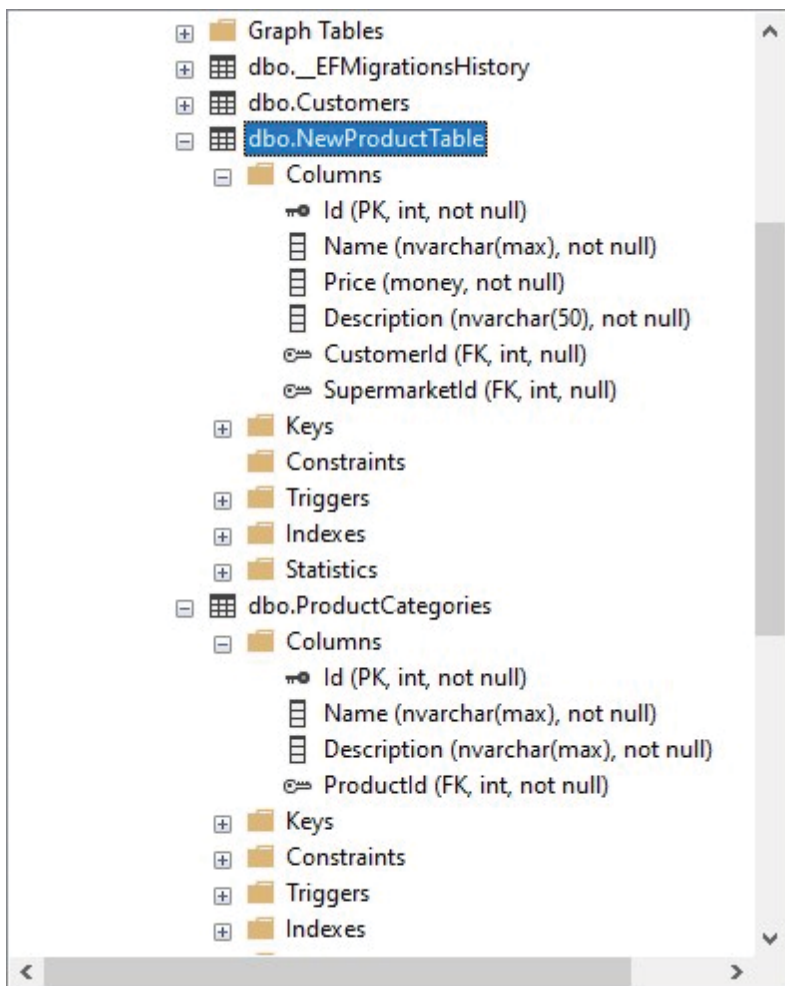


Figure 3.9: Database reflecting the changes made in the code

In a one-to-many relationship, the "one" end of the relationship, represented by the main entity as **Product**, should have one or more primary or alternate key properties such as **Product.Id**. On the other hand, the "many" of the relationship, represented by the dependent entity like **ProductCategory**, should have one or more foreign key properties such as **ProductCategory.ProductId**.

Optionally, the main entity can have a collection navigation like **Product.ProductCategories** to reference the dependent entities, while the dependent entity can have an optional reference

navigation such as `ProductCategory.Product` to reference the principal entity. These components define and establish a one-to-many relationship within Entity Framework Core.

Optional one-to-many

In this scenario, a Product is associated from zero until `n` `ProductCategory`:

```
1. [Table("NewProductTable")]
2.     public class Product
3.     {
4.         public int Id { get; set; }
5.         public string Name { get; set; }
6.         [Column(TypeName = "money")]
7.         public double Price { get; set; }
8.         [MaxLength(50, ErrorMessage = "Max
length is 50!")]
9.         public string Description { get;
set; }
10.     public ICollection<ProductCategory>
Category { get; } = new
List<ProductCategory>();
11.     }
12.     public class ProductCategory
13.     {
14.         public int Id { get; set; }
15.         public string Name { get; set; }
16.         public string Description { get;
set; }
17.         public int? ProductId { get; set; }
18.         public Product? Product { get; set;
} = null!;
19.     }
```

After running the “**add-migration**” command followed by the “**update-database**” command, we have those changes in the database:

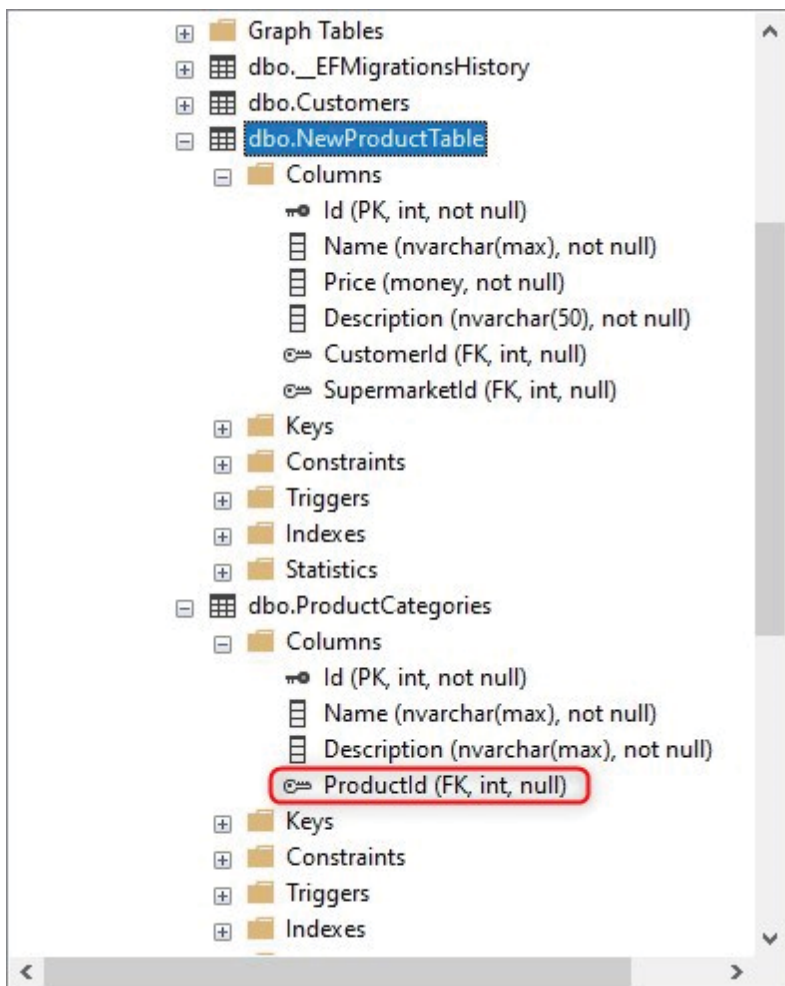


Figure 3.10: Database reflecting the changes made in the code

In an optional one-to-many relationship, we have nullable navigation properties in the many entity of the relationship. We have the `ProductCategory.ProductId`, and `ProductCategory.Product` nullable

Many-to-many relationship

A **many-to-many relationship** represents a scenario where multiple instances of one entity are associated with multiple instances of another entity. To facilitate this relationship, an intermediate table is used to keep the relationship between the

entities.

Entity Framework Core can handle the management of this relationship transparently, allowing the navigations of the many-to-many relationship to be utilized naturally, with the ability to add or remove entities from either side as required. Nevertheless, understanding what happens in the background, the underlying mechanics, is beneficial to take the best out of the overall behavior, especially in terms of the mapping to a relational database.

In this scenario, it is required that many Products are associated with many **ProductCategories**. We are creating a class **Category** and updating **Product** and **ProductCategory**:

```
1. [Table("NewProductTable")]
2. public class Product
3. {
4.     public int Id { get; set; }
5.     public string Name { get; set; }
6.     [Column(TypeName = "money")]
7.     public double Price { get; set; }
8.     [MaxLength(50, ErrorMessage = "Max
length is 50!")]
9.     public string Description { get;
set; }
10.     public List<ProductCategory>
ProductCategories { get; } = new
List<ProductCategory>();
11. }
12.
13. public class Supermarket
14. {
15.     public int Id { get; set; }
16.     public string Address { get; set; }
17.     [Required(ErrorMessage = "The City
is required.")]
18.     public string City { get; set; }
19.     public ICollection<Product>
Products { get; set; }
```

```
20.     }
21.
22.     public class ProductCategory
23.     {
24.         public int Id { get; set; }
25.         public int? ProductId { get; set; }
26.         public int? CategoryId { get; set; }
27.     }
28.     public Product? Product { get; set; }
29.     } = null!;
```

After running the “**add-migration**” command followed by the “**update-database**” command, we have those changes in the database:

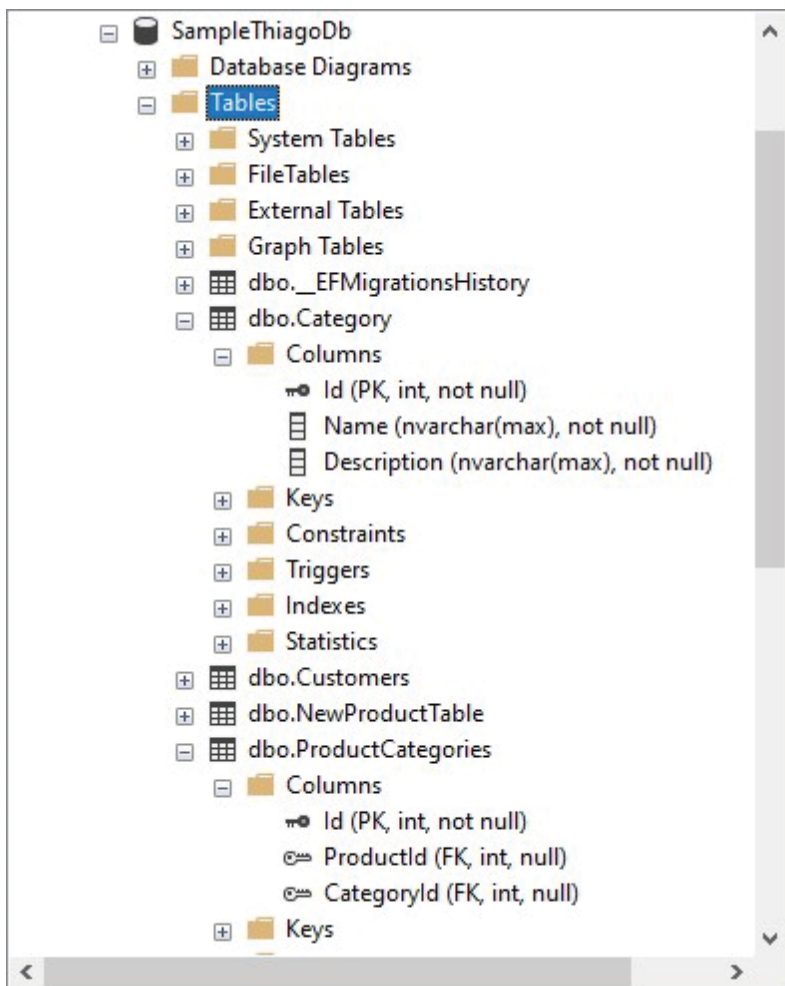


Figure 3.11: Database reflecting the changes made in the code

Data Management

Data Management plays a crucial role in any software application, and Entity Framework Core provides a powerful toolset for managing data access and manipulation. We will explore the key concepts of data management in Entity Framework Core, focusing on three important aspects: normal usage, unit of work, and the repository pattern.

For the following practical examples, we are continuing the project

that we created for the Code First approach and modified for the Data Modeling topic.

Normal usage

Normal usage refers to the common practices and techniques used to interact with the database using Entity Framework Core. It involves performing basic **Create, Read, Update, Delete (CRUD)** operations on entities, querying the database using **Language-Integrated Query (LINQ)**, and handling relationships between entities.

The **DbContext** class in Entity Framework Core acts as a bridge between the application and the database. It represents a session with the database and provides a set of APIs to perform database operations. Developers can create a **DbContext** subclass specific to their application's needs and define **DbSet** properties to represent the entity sets.

Entity Framework Core simplifies the execution of CRUD operations by providing methods like **Add**, **Remove**, **Update**, and **SaveChanges**. These methods abstract the underlying SQL statements required to perform the respective operations, making it easier for developers to work with data.

A new Web API Controller was created to represent the normal usage of Entity Framework Core and the DbContext, we have a CRUD on the Customer entity, as we can see from the following code block:

```
1. [Route("api/[controller]")]
2. [ApiController]
3. public class
   CustomersNormalEFController :
   ControllerBase
4. {
5.     private readonly SampleDbContext
       sampleDbContext;
6.     public
       CustomersNormalEFController(SampleDbContext
       dbContext)
7.     {
```

```
8.             sampleDbContext = dbContext;
9.         }
10.
11.         [HttpGet("Customers/{id}")]
12.         public IActionResult GetById(int
            id)
13.         {
14.             return new
                OkObjectResult(sampleDbContext.Customers.FirstOrDefault
                (x => x.Id == id));
15.         }
16.         [HttpGet("Customers")]
17.         public IActionResult GetAll()
18.         {
19.             return
                Ok(sampleDbContext.Customers.ToList());
20.         }
21.         [HttpPost("Customer")]
22.         public IActionResult
            PostSingle([FromBody] Customer customer)
23.         {
24.             sampleDbContext.Customers.Add(customer);
25.             sampleDbContext.SaveChanges();
26.
27.             return
                CreatedAtAction(nameof(GetById), new { id =
                customer.Id }, customer);
28.         }
29.         [HttpPut("Customer")]
30.         public IActionResult
            PutSingle([FromBody] Customer customer)
31.         {
32.             if (!
                sampleDbContext.Customers.Any(x => x.Id ==
                customer.Id))
33.                 return NotFound();
```



```

34.
35.
36.     sampleDbContext.Customers.Update(customer);
37.     sampleDbContext.SaveChanges();
38.
39.     return NoContent();
40. }
41. [HttpDelete("Customers/{id}")]
42.     public IActionResult Delete(int id)
43.     {
44.         var obj =
sampleDbContext.Customers.Include(x=>x.Products)
=> x.Id == id);
45.         if (obj ==null)
46.             return NotFound();
47.
48.     sampleDbContext.Products.RemoveRange(obj.Products);
49.     sampleDbContext.Customers.Remove(obj);
50.     sampleDbContext.SaveChanges();
51.
52.     return NoContent();
53. }
54. }

```

Unit of work

The **unit of work** pattern is commonly used in software development to manage transactions and ensure atomicity and consistency when working with data. In Entity Framework Core, the unit of work pattern can be implemented using the **DbContext** class and its underlying change tracking mechanism.

Entity Framework Core tracks changes made to entities within a **DbContext** instance. This means that any modifications, additions, or deletions performed on entities are kept in memory until explicitly saved to the database. The unit of work pattern leverages

this change tracking mechanism to manage transactions effectively.

With the unit of work pattern, developers can group multiple database operations into a single transaction. By using the **BeginTransaction** and **Commit/Rollback** methods of the **DbContext**, changes made to entities can be either committed as a whole or rolled back entirely in case of an error or exception.

A new Web API Controller was created to represent the unit of work with Entity Framework Core, we have a CRUD on the **Customer** entity, as we can see from the following code block:

```
1. [Route("api/[controller]")]
2.     [ApiController]
3.     public class
CustomersUnitOfWorkController :
ControllerBase
4.     {
5.         private readonly UnitOfWork
unitOfWork;
6.         public
CustomersUnitOfWorkController(SampleDbContext
dbContext)
7.         {
8.             unitOfWork = new
UnitOfWork(dbContext);
9.         }
10.        [HttpGet("Customers/{id}")]
11.        public IActionResult GetById(int
id)
12.        {
13.            return new
OkObjectResult(unitOfWork.CustomerRepository.GetI
14.        }
15.        [HttpGet("Customers")]
16.        public IActionResult GetAll()
17.        {
18.            return
Ok(unitOfWork.CustomerRepository.Get());
```

```

19.         }
20.         [HttpPost("Customer")]
21.         public IActionResult
PostSingle([FromBody] Customer customer)
22.         {
23.             unitOfWork.CustomerRepository.Insert(customer);
24.             unitOfWork.Save();
25.
26.             return
CreatedAtAction(nameof(GetById), new { id =
customer.Id }, customer);
27.         }
28.         [HttpPut("Customer")]
29.         public IActionResult
PutSingle([FromBody] Customer customer)
30.         {
31.             unitOfWork.CustomerRepository.Update(customer);
32.             unitOfWork.Save();
33.
34.             return NoContent();
35.         }
36.         [HttpDelete("Customers/{id}")]
37.         public IActionResult Delete(int id)
38.         {
39.             var customer =
unitOfWork.CustomerRepository.Get(x =>
x.Id.Equals(id), includeProperties:
"Products").FirstOrDefault();
40.             foreach (var item in
customer.Products)
41.                 unitOfWork.ProductRepository.Delete(item.Id);
42.
43.

```

```

unitOfWork.CustomerRepository.Delete(id);
44.         unitOfWork.Save();
45.
46.         return NoContent();
47.     }
48. }

```

Repository pattern

The **repository pattern** is a design pattern that provides an abstraction layer between the application and the data access layer. It helps decouple the application from the specific data access technology, such as Entity Framework Core, by defining a set of interfaces and classes representing data repositories.

In the repository pattern, interfaces are defined to specify the contract for data access operations. These interfaces typically include methods for CRUD operations, querying, and any other specific data access requirements of the application.

Concrete implementations of repository interfaces are created to interact with the underlying data store, which in this case is Entity Framework Core. These implementations utilize the **DbContext** and its **DbSet** properties to perform the required operations.

The repository pattern promotes the separation of concerns by isolating the data access logic from the rest of the application. This enables better testability, maintainability, and flexibility in switching between different data access technologies.

A new Web API Controller was created to represent the repository pattern of Entity Framework Core, we have a CRUD on the Customer entity, as we can see from the following code block:

```

1. [Route("api/[controller]")]
2. [ApiController]
3. public class
CustomersRepoPatternController :
ControllerBase
4. {
5.     private readonly
GenericRepository<Customer>

```

```

customerRepository;

6.         private readonly
GenericRepository<Product>
productRepository;

7.         public
CustomersRepoPatternController (GenericRepository<Product>
customerRepository,
GenericRepository<Product>
productRepository)

8.         {

9.             this.customerRepository =
customerRepository;

10.            this.productRepository =
productRepository;

11.        }

12.

13.        [HttpGet ("Customers/{id}")]

14.        public IActionResult GetById (int
id)

15.        {

16.            return new
OkObjectResult (customerRepository.GetById (id));

17.        }

18.        [HttpGet ("Customers")]

19.        public IActionResult GetAll ()

20.        {

21.            return
Ok (customerRepository.Get ());

22.        }

23.        [HttpPost ("Customer")]

24.        public IActionResult
PostSingle ([FromBody] Customer customer)

25.        {

26.            customerRepository.Insert (customer);

27.            customerRepository.context.SaveChanges ();

```

```

28.
29.         return
    CreatedAtAction(nameof(GetById), new { id =
    customer.Id }, customer);
30.     }
31.     [HttpPut("Customer")]
32.     public IActionResult
    PutSingle([FromBody] Customer customer)
33.     {
34.
    customerRepository.Update(customer);
35.
    customerRepository.context.SaveChanges();
36.
37.         return NoContent();
38.     }
39.     [HttpDelete("Customers/{id}")]
40.     public IActionResult Delete(int id)
41.     {
42.         var customer =
    customerRepository.Get(x => x.Id.Equals(
    id), includeProperties:
    "Products").FirstOrDefault();
43.         foreach (var item in
    customer.Products)
44.
    productRepository.Delete(item.Id);
45.
    customerRepository.Delete(id);
46.
    customerRepository.context.SaveChanges();
47.
48.
49.         return NoContent();
50.     }
51. }

```

A few changes were made in the **Program.cs** to Inject the dependencies applied in the examples above. You can see the full

Program.cs below:

```
1. using Microsoft.EntityFrameworkCore;
2. using WebAppEFCoreCodeFirst.Database;
3.
4. var builder =
    WebApplication.CreateBuilder(args);
5. builder.Services.AddMvc(options =>
    options.EnableEndpointRouting = false);
6. builder.Services.AddScoped<GenericRepository<Customer>>();
7. builder.Services.AddScoped<GenericRepository<Product>>();
8. // Add services to the container.
9. builder.Services.AddRazorPages();
10. builder.Services.AddDbContext<SampleDbContext>(
11.     options =>
        options.UseSqlServer("Data Source=DESKTOP-
        H20012E;Initial
        Catalog=SampleThiagoDb;Integrated
        Security=True;Connect
        Timeout=30;Encrypt=False;Trust Server
        Certificate=False;Application
        Intent=ReadWrite;Multi Subnet
        Failover=False"));
12.
13. var app = builder.Build();
14.
15. // Configure the HTTP request pipeline.
16. if (!app.Environment.IsDevelopment())
17. {
18.     app.UseExceptionHandler("/Error");
19.     // The default HSTS value is 30 days. You may want to
        change this for production scenarios, see https://aka.ms/
        aspnetcore-hsts.
20.     app.UseHsts();
21. }
22.
23.
```

```
24. app.UseHttpsRedirection();
25. app.UseStaticFiles();
26. app.UseMvc();
27. app.UseRouting();
28.
29. app.UseAuthorization();
30.
31. app.MapRazorPages();
32.
33. app.Run();
```

Conclusion

In conclusion, this chapter has provided a comprehensive exploration of **Entity Framework Core (EF Core)** and its essential components, including Database First, Code First, LINQ to Entities, Data Annotations, Data Modeling, and Data Management. Throughout this chapter, we have delved into the various approaches and techniques to master EF Core.

We began by understanding the Database First approach, which enables us to generate entity classes and the DbContext from an existing database schema. We explored customization options, allowing us to tailor the generated code to our specific needs. This approach is particularly useful when working with legacy databases or integrating EF Core into an existing project. Next, we discussed the Code First approach, where we learned how to define entity classes, configure relationships, and generate the database schema using migrations. This approach provides flexibility and control over the database design, making it ideal for greenfield projects or scenarios where the database schema is developed alongside the application. We then dived into the power of LINQ to Entities, which empowers us to write expressive queries and perform advanced data manipulation within EF Core. By leveraging LINQ's rich query syntax and operators, we can efficiently retrieve, filter, and shape data to meet our application's requirements.

Data Annotations proved to be an invaluable tool in our arsenal, enabling us to apply validation rules to entities, map them to database tables, and define relationships. With Data Annotations,

we can ensure data integrity, streamline data access, and improve the overall efficiency of our application.

We also explored the realm of Data Modeling in EF Core, encompassing entity relationships, inheritance, and complex data types. By mastering these modeling techniques, we can design robust and scalable data models that accurately represent the domain and enable efficient data management. Lastly, we discussed Data Management, which covered various aspects such as efficient data retrieval, modification, deletion, and optimization techniques. We explored strategies to handle transactions, concurrency control, and effectively manage large datasets. These skills are essential for building high-performing applications that can handle complex data scenarios with ease.

By accomplishing the objectives outlined in this chapter, you have acquired a solid foundation in mastering Entity Framework Core. Whether you prefer the Database First or Code First approach, have gained proficiency in using LINQ to Entities for data manipulation, understand the benefits of Data Annotations, can effectively design and manage data models, and are equipped with the skills to optimize data operations.

With this newfound knowledge, you are well-prepared to harness the power of Entity Framework Core and build robust, scalable, and efficient data access layers in your applications. EF Core's versatility and feature-rich ecosystem will empower you to tackle even the most challenging data scenarios, ensuring the success of your projects.

In the upcoming chapter we will explore serverless computing with Azure Functions. This chapter serves as a comprehensive guide to understanding the fundamental concepts of Azure Functions, beginning with an introduction to Triggers and Bindings. Through a step-by-step case study, you will learn how to create Azure Functions, select appropriate triggers, and leverage bindings to seamlessly integrate with other Azure services. By the end of this chapter, you will be equipped with the knowledge and practical skills to build and deploy your own Azure Functions, enabling you to implement scalable and efficient cloud-based solutions.

CHAPTER 4

Getting Started with Azure Functions

Introduction

Azure Functions is a powerful serverless computing service provided by Microsoft Azure that allows you to build and deploy applications quickly and efficiently. It enables you to focus on writing code for specific tasks or functionalities without worrying about managing the underlying infrastructure. With Azure Functions, you can create small, single-purpose functions that respond to events and scale automatically to handle varying workloads.

In this chapter, we will explore the fundamentals of Azure Functions and guide you through getting started with this versatile service. We will begin by clearly defining Azure Functions and highlighting their key features and benefits. Understanding these foundational concepts will help you grasp the true potential of Azure Functions and how they can streamline your application development and deployment.

Next, we will delve into triggers and bindings, which are essential components of Azure Functions. Triggers are the events that initiate the execution of your functions, while bindings facilitate seamless

integration with various data sources and services.

Additionally, we will cover the concept of bindings and their role in connecting your functions to external resources processing input data and generating output from your functions simple.

We will present a case study with a practical example to reinforce the concepts discussed. We will walk you through a real-world scenario, guiding you in creating an Azure Function that demonstrates the power and versatility of this serverless computing service. From selecting the appropriate trigger to choosing the right binding and testing the output, you will gain hands-on experience in building and deploying an Azure Function.

So, let us dive in and get started with Azure Functions!

Structure

This chapter covers the following topics:

- Getting started with Azure Functions
- Azure Function triggers
- Azure Function bindings
- Practical case-study
 - Creating the Azure Function
 - Selecting the trigger
 - Picking an appropriate binding
 - Testing the output

Objectives

Throughout this chapter, we aim to provide a comprehensive understanding of Azure Functions and their practical implementation. We will begin by exploring the core concepts and purposes of Azure Functions, shedding light on their relevance in modern application development. Furthermore, we will delve into the key features and benefits of Azure Functions, helping you grasp their significance in building scalable and efficient solutions.

In our exploration, we will conduct a comparative analysis of Azure Functions against other serverless offerings available in the market.

We will also explain Azure Function bindings, which serve as the bridges connecting your functions to external resources and data.

We will explore the scaling options available for Azure Functions and delve into the monitoring capabilities, helping you optimize the performance of your functions.

As our chapter concludes, we will guide you through the deployment process, demonstrating how to deploy Azure Functions to Azure. Additionally, you'll learn about implementing CI/CD pipelines for continuous deployment, ensuring a seamless and efficient deployment workflow.

Getting started with Azure Functions

Azure Functions is a serverless computing service provided by Microsoft Azure that enables developers to build and deploy small, event-driven functions in the cloud. It allows you to focus on writing code for specific tasks or functionalities without the need to manage the underlying infrastructure. Azure Functions automatically scales and allocates resources based on the workload, ensuring efficient utilization and cost-effectiveness.

Key features and benefits of Azure Functions include:

- **Serverless architecture:** Azure Functions follows a serverless architecture, where you write and deploy code as individual functions, and the cloud provider manages the infrastructure and resource allocation. You are charged only for the execution time of your functions, eliminating the need to provision and manage servers.
- **Event-driven execution:** Azure Functions are triggered by events, such as HTTP requests, timers, messages in queues, or changes in Azure Storage. Each function is designed to respond to a specific event and execute a defined action or process.
- **Multiple language support:** Azure Functions supports various programming languages, including C#, JavaScript, Python, PowerShell, and TypeScript. This allows you to choose the language that best fits your development preferences and existing codebase.
- **Integration and bindings:** Azure Functions provide seamless integration with various Azure services and external systems

through bindings. Bindings allow you to connect functions to data sources, message queues, storage services, and other resources, simplifying input and output operations.

- **Scalability and elasticity:** Azure Functions automatically scales based on the incoming workload, allowing your applications to handle high traffic and spikes in demand without manual intervention. Functions are executed in parallel and can be horizontally scaled to accommodate varying workloads.
- **Pay-as-you-go pricing:** Azure Functions offer a pay-per-use pricing model, where you are charged based on the actual execution time and resources consumed by your functions. This makes it cost-effective for applications with varying or unpredictable workloads.
- **Development and deployment options:** Azure Functions provide multiple options for development and deployment, including Azure Portal, Visual Studio, Visual Studio Code, Azure DevOps, and Azure Functions Core Tools. This flexibility allows developers to choose their preferred environment and streamline the development and deployment.
- **Monitoring and diagnostics:** Azure Functions offer built-in monitoring and diagnostics capabilities through Azure Application Insights. You can monitor the performance, availability, and usage of your functions, as well as set up alerts and view detailed logs for troubleshooting and optimization.
- **DevOps integration:** Azure Functions seamlessly integrate with Azure DevOps, enabling **Continuous Integration and Continuous Deployment (CI/CD)** pipelines. This ensures smooth deployment and updates of your functions, allowing for rapid development and iteration.

Azure Functions provide a flexible and scalable platform for building various applications, from simple microservices to complex event-driven architectures. With its serverless nature, wide language support, and deep integration with other Azure Services, Azure Functions empower developers to focus on code logic and deliver scalable and efficient solutions.

Azure Function triggers

Azure Functions are triggered by events, which initiate the execution of a specific function. Triggers define when and how a function should run. Azure Functions support various triggers, allowing you to build event-driven applications that respond to different types of events.

Here are some commonly used Azure Function triggers:

- **HTTP trigger:** An HTTP trigger allows you to invoke a function via an HTTP request. This trigger is useful for building RESTful APIs or handling webhooks.
- **Timer trigger:** A timer trigger allows you to schedule the execution of a function at specified time intervals or specific times. This trigger is ideal for running periodic background tasks or performing scheduled data processing.
- **Queue trigger:** A queue trigger enables the function to be triggered when a message is added to a message queue, such as Azure Storage queue or Azure Service bus queue. This trigger is often used for building asynchronous processing workflows or decoupled systems.
- **Blob trigger:** A blob trigger executes the function when a new or updated blob is detected in Azure Blob storage. This trigger is commonly used to process files, perform image or data transformations, or trigger data ingestion pipelines.
- **Event Grid trigger:** An event grid trigger allows you to respond to events from various Azure Services or custom events published to the Azure Event Grid. This trigger provides a flexible and centralized event routing mechanism.
- **Cosmos DB trigger:** A Cosmos DB trigger executes the function when changes occur in Azure Cosmos DB collections. This trigger is useful for real-time processing of data updates, notifications, or synchronization tasks.
- **Service bus trigger:** A service bus trigger triggers the function when a new message arrives in an Azure Service bus topic or subscription. This trigger is often used for building publish-subscribe messaging patterns or integrating with enterprise messaging systems.
- **Event hub trigger:** An event hub trigger allows the function to be triggered by events received from Azure Event Hub. This trigger is suitable for processing high-throughput streaming data or building real-time analytics solutions.

These are just a few examples of the triggers available in Azure Functions. Each trigger type has its own properties and configuration options, allowing you to customize the behavior of your functions based on the event source and specific application requirements. By leveraging these triggers, you can build responsive and event-driven applications that seamlessly react to various events in your system or external services.

Azure Function bindings

Azure Function bindings are a powerful feature that enables seamless integration with external resources and services. Bindings act as connectors between functions and the data sources, message queues, storage services, and other resources that your functions interact with.

Here are some commonly used Azure Function bindings:

- **Input bindings:** Input bindings allow functions to receive data from external sources. Some examples include:
 - **Blob storage binding:** Allows the function to read input data from or write output data to Azure Blob storage.
 - **Cosmos DB binding:** Enables reading or writing data to Azure Cosmos DB collections.
 - **HTTP binding:** Allows the function to receive data from an HTTP request and extract information from the request's headers, body, or query parameters.
 - **Queue storage binding:** Enables receiving and processing messages from Azure Storage queues.
 - **Event grid binding:** Enables subscribing to events from Azure Event Grid and triggering the function when specific events occur.
- **Output bindings:** Output bindings enable functions to send data to external resources. Some examples include:
 - **Blob storage binding:** Enables the function to write output data to Azure Blob storage.
 - **Cosmos DB binding:** Allows the function to store or update data in Azure Cosmos DB collections.
 - **HTTP binding:** Enables the function to send HTTP

responses or make HTTP requests to external APIs or services.

- **Queue storage binding:** Enables the function to add messages to Azure Storage queues.
- **Event hub binding:** Allows the function to send events to Azure Event Hubs for real-time streaming or analytics scenarios.
- **Trigger bindings:** Trigger bindings define the event or condition that initiates the execution of a function. Some examples include:
 - **Timer trigger binding:** Schedules the function to run at specified time intervals or specific times.
 - **HTTP trigger binding:** Triggers the function execution when an HTTP request is received.
 - **Queue trigger binding:** Initiates the function execution when a new message is added to an Azure Storage queue.
 - **Cosmos DB trigger binding:** Triggers the function when changes occur in Azure Cosmos DB collections.
 - **Event Grid trigger binding:** Triggers the function when specific events are published to the Azure Event Grid.

Bindings provide a declarative way to configure and manage the interaction between your functions and external resources. They eliminate the need for manual integration code and simplify the development and maintenance of Azure Functions. By leveraging bindings, you can easily integrate your functions with various data sources, services, and event publishers, enabling seamless data processing, storage, and communication within your applications.

Practical case-study

In this case study, we will create an Azure Function with an HTTP trigger, and we will configure two output bindings: one for creating a Blob in an Azure Storage Account and another for publishing an event to Azure Event Hub. This setup involves a single trigger and two output bindings, enabling us to seamlessly handle incoming HTTP requests and store data in Blob storage while simultaneously triggering events for further processing or analysis through Azure Event Hub. By leveraging these bindings, we can easily integrate

our Azure Function with multiple services, enabling efficient data processing and event-driven workflows.

Creating the Azure Function

To create an Azure Function from the Azure Portal, follow these steps:

1. Sign in to the Azure Portal (**portal.azure.com**) using your credentials.
2. Navigate to the Azure Functions service by either searching for "**Functions**" in the search bar or locating it in the service list.
3. Click on **Create Function** or **Add** to initiate the function creation process.
4. In the Function App creation settings, provide the following details:
 - **Function App Name:** Choose a unique name for your Function App.
 - **Subscription:** Select the Azure subscription under which the Function App will be created.
 - **Resource Group:** Specify the resource group to which the Function App will belong.
 - **Operating System:** Choose the operating system for the Function App (Windows or Linux).
 - **Hosting Plan:** Select the appropriate hosting plan, such as Consumption (serverless) or App Service (dedicated).
10. Next, configure the Runtime Stack settings.
 - **Runtime Stack:** Choose the programming language for your function, such as C#, JavaScript, Python, PowerShell, or TypeScript.
12. Configure the Storage Account settings:
 - **Storage Account:** Select an existing storage account or create a new one to store function execution logs and other metadata.
14. Optionally, configure Application Insights:
 - **Application Insights:** Enable or disable Application Insights for monitoring and diagnostics purposes.

16. You may also find additional advanced settings available:
 - **Region:** Specify the Azure region where your Function App will be deployed.
 - **Runtime Version:** Choose the version of the Azure Functions runtime.
 - **Authentication:** Configure authentication options for your Function App.
 - **Networking:** Configure networking settings, such as Virtual Network integration or Private Endpoint.
21. Once you have provided all the necessary settings, review your configuration and click on **Create** to initiate the creation of your Azure Function.

You should see the following confirmation screen:

Create Function App ...



Details

Subscription	b184a6c9-1f65-4ad3-b46b-1d09053ac048
Resource Group	functionAppRH
Name	testthiago
Runtime stack	.NET 6 (LTS)

Hosting

Plan (New)

Hosting options and plans	Consumption (Serverless)
Name	ASP-functionAppRH-83c0
Operating System	Windows
Region	West Europe
SKU	Dynamic

Monitoring (New)

Application Insights	Enabled
Name	testthiago
Region	West Europe

Create

< Previous

Next >

[Download a template for automation](#)

Figure 4.1: Azure function settings before creation.

After a successful deployment, you will be presented with the Azure Function Overview, which provides a comprehensive view of your newly created Azure Function and its associated resources. The Azure Function Overview offers valuable information and functionality for managing and monitoring your function.

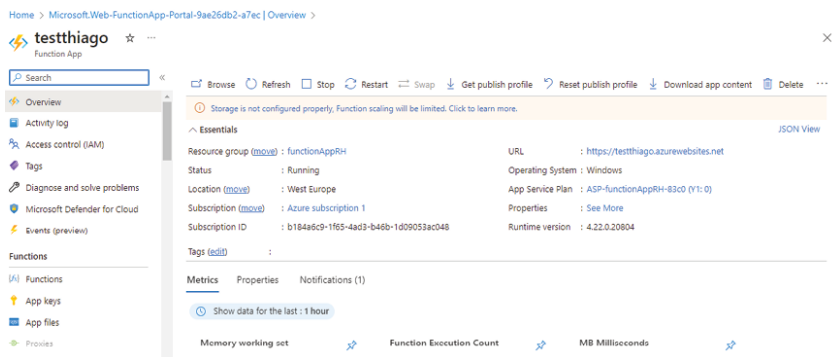


Figure 4.2: Azure Function overview.

Selecting the trigger

Now, let us proceed with creating the function by following these steps:

1. The first step is to select the appropriate trigger type. The trigger determines the event or condition that will initiate the execution of your function. It defines when your function will run and what will trigger its execution.
2. You will be presented with a list of available templates. Select the template that corresponds to the desired trigger type. For example, if you want to trigger the function based on an HTTP request, choose the **HTTP trigger** template.
3. Configure the trigger properties, such as the route or URL path for HTTP triggers or the schedule for timer triggers. Customize the settings according to your application requirements.

Optionally, provide additional details such as the authentication level for HTTP triggers or the name and direction of the binding for input/output triggers.

1. Once you have configured the trigger, click on the **Create** or **Save** button to create the function with the selected trigger. You should see the following screen proceeding the function creation. Push **Create** to continue:

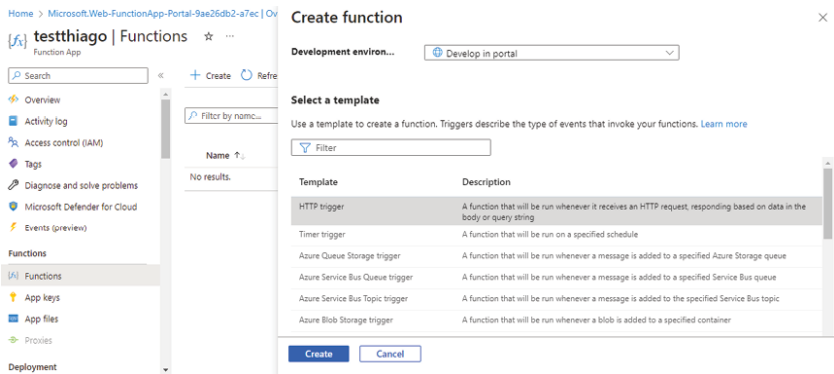


Figure 4.3: Selecting the trigger for the Azure Function.

After creating the function, you will be presented with the output, indicating that your function has been successfully created. At this point, your function is ready to run, and you will have a boilerplate code provided as a starting point. This code serves as a foundation for implementing the logic specific to your function's requirements.

With the boilerplate code provided, you have a solid starting point for your function. You can modify and enhance the code to meet your specific requirements, incorporating the necessary business logic and interacting with external resources through bindings. Take advantage of this foundation to quickly build and deploy your custom function while leveraging the benefits of Azure Functions. You can see a Azure Function running on Azure Portal below:

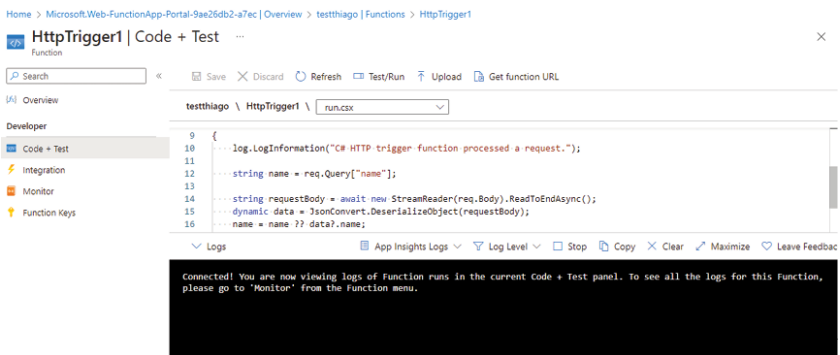


Figure 4.4: Output for the created function with the HTTP trigger.

Picking an appropriate binding

Now, let us move on to setting up our output bindings, starting with the Storage Account. Once you have configured your Azure Storage Account, you are ready to set up the output binding in your Azure Function. The output binding will allow your function to create and store data in the Blob Container within the Storage Account. Follow the next steps to configure the output binding and establish the connection with the Storage Account, here is my storage account created for this example:

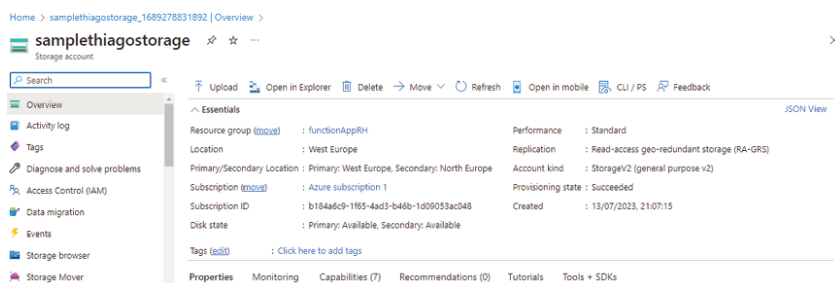


Figure 4.5: The recently created Storage Account overview.

Before proceeding further, ensure that you have created an Azure Event Hub namespace. You can see an example of an Azure Event Hub namespace below:

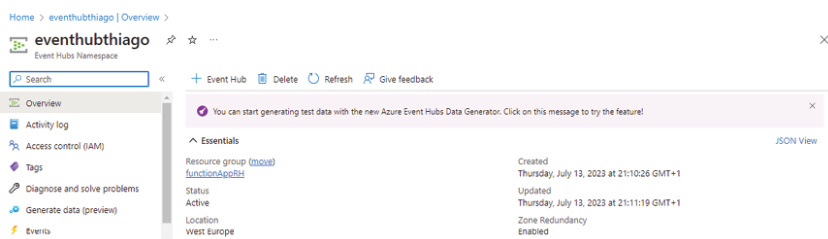


Figure 4.6: The recently created Event Hub Namespace overview.

Additionally, ensure that your Azure Event Hub namespace has an Event Hub created within it. You can see an example of an Azure Event Hub below:

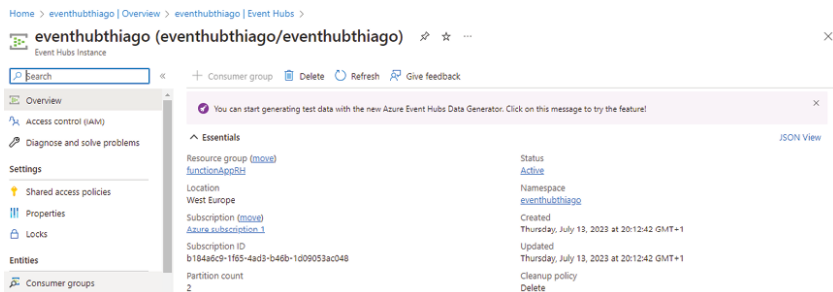


Figure 4.7: The recently created Event Hub, inside the Event Hub Namespace.

Now, let us shift our focus back to the Azure Function, as it is time to configure the output bindings. Configuring the output bindings allows your function to seamlessly interact with external resources and services as you can see from the figure below:

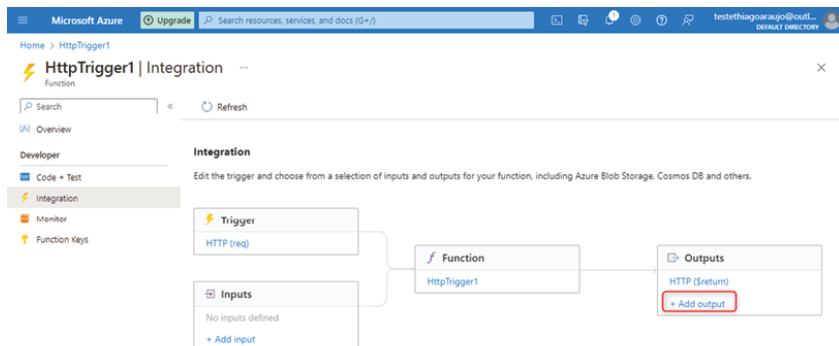


Figure 4.8: Adding more output for the Azure Function.

Let us begin by setting up our Blob Storage binding. This binding will allow your Azure Function to interact with Azure Blob storage. You may check how to set the output from the picture below:

Create Output ×

Start by selecting the type of output binding you want to add.

Binding Type

Azure Blob Storage ▼

Azure Blob Storage details

Storage account connection * ⓘ

samplethiagostorage_STORAGE (new) ▼

[New](#)

Blob parameter name * ⓘ

outputBlob

Path * ⓘ

OK

Cancel

Figure 4.9: *Configuring the Azure Blob Storage as output.*

Now, let us proceed to set up our second output binding, the Event Hub. This binding will enable your Azure Function to publish events or messages to an Azure Event Hub. You may check how to set the second output from the picture below:

Create Output

×

Start by selecting the type of output binding you want to add.

Binding Type

Azure Event Hubs

Azure Event Hubs details

Event Hub connection * ⓘ

eventhubthiago_RootManageShared... ▾

New

Event parameter name * ⓘ

outputEventHubMessage

OKCancel

Figure 4.10: *Configuring the Event Hub as output.*

In this example, we have configured the Event Hub to register captures in a Storage Account:

Home > eventhubthiago | Event Hubs > eventhubthiago (eventhubthiago/eventhubthiago) > eventhubthiago | Event Hubs > eventhubthiago (eventhubthiago/eventhubthiago)

eventhubthiago (eventhubthiago/eventhubthiago) | Capture ☆ ...

Event Hubs Instance

Search

Save changes ✕ Discard 🔄 Give feedback

Overview

Access control (IAM)

Diagnose and solve problems

Settings

Shared access policies

Properties

Locks

Entities

Consumer groups

Features

...

Capture Provider

Azure Storage Account

Azure Storage Container *

eventhubcontainer ▾

Select Container

Storage Account

/subscriptions/b184a6c9-1f65-4ad3-b46b-1d0903ac048/resourceGroups/functionAppRt/providers/Microsoft.Storage/storageAccounts/samplethiagostorage

Sample Capture file name formats

[Namespace]/(EventHub)/(PartitionId)/(Year)/(Month)/(Day)/(Hour)/(Minute)/(Second)

Capture file name format ⓘ

[Namespace]/(EventHub)/(PartitionId)/(Year)/(Month)/(Day)/(Hour)/(Minute)/(Second)

e.g. eventhubthiago/eventhubthiago/0/2023/07/13/20/43/08

Authentication for Capture

Figure 4.11: *Overview of the capture settings from the Event Hub.*

After successfully setting up both output bindings, your Azure Function should resemble the following structure, as shown in [Figure 4.12](#):

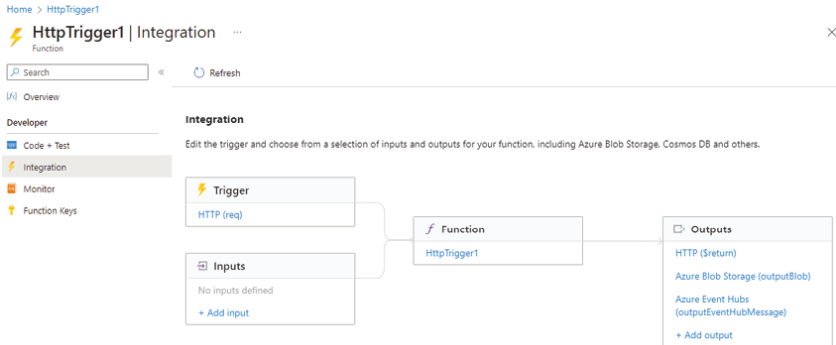


Figure 4.12: The Azure Function flow after adding both outputs.

It is time to update our code so that we can create the Blob and publish the Event. Here is an example of how your **Run.csx** file should look like:

```
1. #r "Newtonsoft.Json"
2.
3. using System.Net;
4. using Microsoft.AspNetCore.Mvc;
5. using Microsoft.Extensions.Primitives;
6. using Newtonsoft.Json;
7.
8. public static IActionResult Run(HttpRequest req, ILogger log, out string outputBlob,
9. out string outputEventHubMessage)
10. {
11.     log.LogInformation("C# HTTP trigger function processed a request.");
12.
13.     string name = req.Query["name"];
14.
15.     string requestBody = new
16.     StreamReader(req.Body).ReadToEnd();
17.
18.     dynamic data =
```

```

    JsonConvert.DeserializeObject(requestBody);
16.     name = name ?? data?.name;
17.
18.     string responseMessage =
        string.IsNullOrEmpty(name)
19.         ? "This HTTP triggered function
            executed successfully. Pass a name in the
            query string or in the request body for a
            personalized response."
20.         : $"Hello, {name}. This
            HTTP triggered function executed
            successfully.";
21.
22.     outputBlob= "Blob:" +name;
23.     outputEventHubMessage= "Message by:
        "+name;
24.     return new
        OkObjectResult(responseMessage);
25. }

```

Here is an example of how your **function.json** file may look like, which is automatically generated based on the bindings you have configured:

```

1. {
2.   "bindings": [
3.     {
4.       "authLevel": "function",
5.       "name": "req",
6.       "type": "httpTrigger",
7.       "direction": "in",
8.       "methods": [
9.         "get",
10.        "post"
11.      ]
12.    },
13.    {
14.      "name": "$return",

```

```

15.     "type": "http",
16.     "direction": "out"
17. },
18. {
19.     "name": "outputBlob",
20.     "direction": "out",
21.     "type": "blob",
22.     "connection":
23.         "samplethiagostorage_STORAGE",
24.     "path": "outcontainer/{rand-guid}"
25. },
26. {
27.     "name": "outputEventHubMessage",
28.     "direction": "out",
29.     "type": "eventHub",
30.     "connection":
31.         "eventhubthiago_RootManageSharedAccessKey_EVENTHUB",
32.     "eventName": "outeventhub"
33. }

```

In the `function.json` file:

- The `bindings` property contains an array of binding configurations for each input and output binding.
- The HTTP trigger binding is defined with the `"type": "httpTrigger"` and `"direction": "in"` properties, specifying the supported HTTP methods (`"methods": ["get", "post"]`) and the name of the trigger (`"name": "req"`).
- The Blob Storage output binding is defined with the `"type": "blob"` and `"direction": "out"` properties. It includes the Blob container path (`"path": "containerName/{blobName}"`), the connection string (`"connection": "<connectionString>"`), and the data type (`"dataType": "string"`).
- The Event Hub output binding is defined with the `"type": "eventHub"` and `"direction": "out"` properties. It includes

the Event Hub name (`"eventHubName": "<eventHubName>"`), the connection string (`"connection": "<connectionString>"`), and the data type (`"dataType": "string"`).

- The `"scriptFile"` property specifies the location of the function's compiled code (`"../bin/MyFunction.dll"`).
- The `"entryPoint"` property defines the function's entry point method (`"MyFunction.Run"`).

Testing the output

We are now ready to run the Azure Function that has been successfully created, and it is time to test its functionality. Follow these steps to test your Azure Function:

1. In the Azure Portal, navigate to your Azure Function within the Function App.
2. In the Function Overview, locate the function you want to test.
3. Open the function's code editor or configuration page.
4. Depending on the trigger type you selected for your function, there are different ways to initiate its execution:
 - **HTTP trigger:** Use a tool like cURL, Postman, or a web browser to send an HTTP request to the function's endpoint. Make sure to provide any required request parameters or body content based on your function's implementation.
 - **Timer trigger:** If you have configured a timer trigger, you can wait for the scheduled time or manually run the function from the function's configuration page.
 - **Other trigger types:** Follow the specific steps or conditions associated with the chosen trigger to initiate the function's execution.
8. Monitor the execution of your function:
 - **Check the logs:** In the Azure Portal, navigate to the Monitoring or Logs section to view the real-time logs and diagnose any issues or obtain information about the function's execution.

- **Verify the output:** If your function produces output such as a Blob in Azure Storage or events published to an Event Hub, ensure the output is generated correctly and matches your expectations.

11. Review the test results and troubleshoot any errors or unexpected behavior that may arise. You may check the output from the Azure Function:

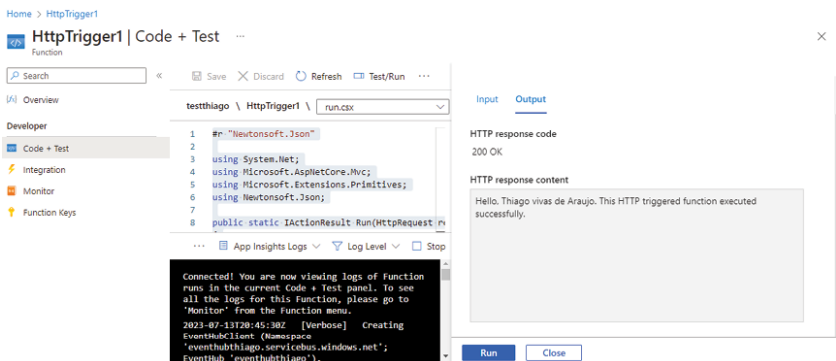


Figure 4.13: Running the Azure Function from the Azure Portal.

We can also trigger the function from the browser, as we can see from the picture below:

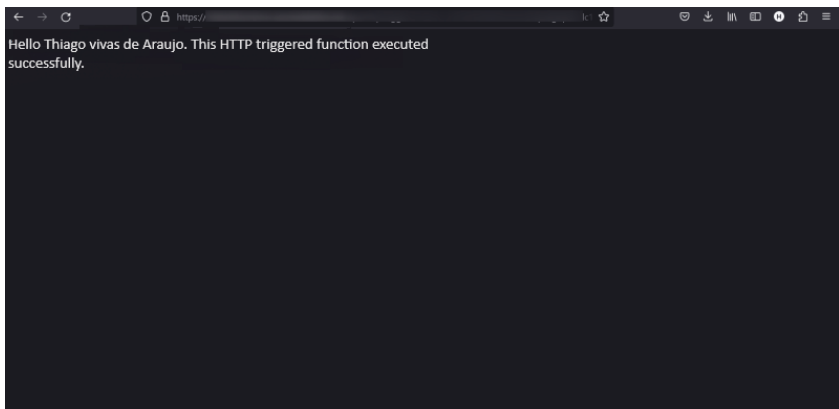


Figure 4.14: Running the Azure Function from the Web Browser.

Let us proceed to validate the output bindings, starting with the Blob Storage. You may check the blob successfully created from the

following figure:

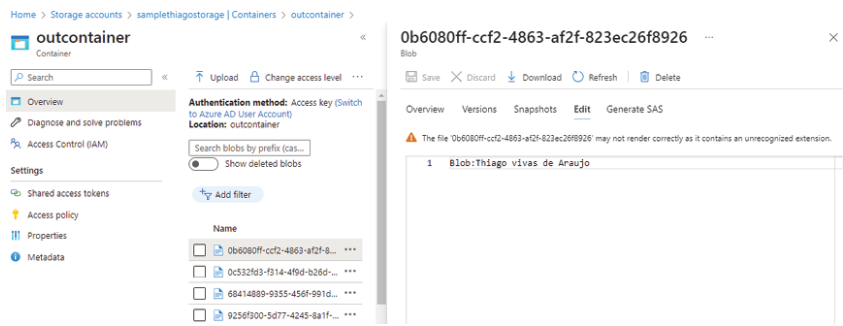


Figure 4.15: Blob Storage Successfully created.

Now, let us check if the Event Hub was successfully created and if events are being published to it. You may see the event created successfully from the picture below:

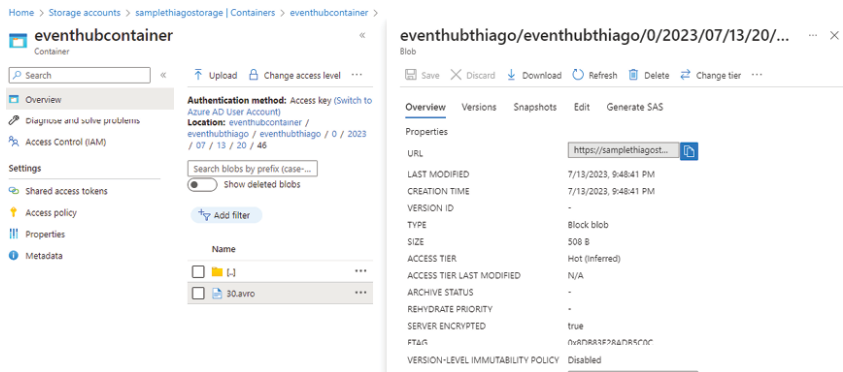


Figure 4.16: Event successfully published to Event Hub.

Conclusion

In this chapter, we have embarked on our journey to explore Azure Functions and get started with this powerful serverless computing service. We began by understanding the concept and purpose of Azure Functions, recognizing their benefits, and comparing them to other serverless offerings. We then delved into the fundamental building blocks of Azure Functions, namely triggers and bindings, which play a crucial role in enabling event-driven application development and seamless integration with external resources.

To solidify our understanding, we dived into a practical case study, where we walked through the process of creating an Azure Function, selecting an appropriate trigger, and configuring the binding. This hands-on experience allowed us to witness firsthand how Azure Functions can be utilized to solve real-world challenges.

Throughout this chapter, we have covered essential topics such as testing the output of Azure Functions, scaling options, and monitoring capabilities. We also touched upon deployment strategies and best practices for managing and monitoring Azure Functions throughout their lifecycle.

By now, you should have a strong foundation in Azure Functions and be equipped with the knowledge and skills necessary to create your own functions. You understand the power and versatility of triggers and bindings and their role in building scalable, event-driven applications.

In the upcoming chapter we'll embark on a journey into the heart of Azure's powerful data management and storage services. Get ready to explore the capabilities of Azure SQL for robust relational data management, dive into the world of globally-distributed NoSQL databases with Azure Cosmos DB, and discover the versatility of Azure Blob Storage for handling unstructured data. This chapter will equip you with the knowledge and practical skills to harness these Azure services to efficiently store, manage, and query your data, taking your cloud-based applications to the next level.

Points to remember

- Azure Functions Fundamentals:
 - Azure Functions are serverless compute resources that allow you to run code in response to various events.
 - They are highly scalable and cost-effective, as you only pay for the resources used during execution.
- Triggers and Bindings:
 - Triggers initiate the execution of Azure Functions based on events or schedules, such as HTTP requests, timers, or queue messages.
 - Bindings facilitate interaction with external resources, allowing input and output data to be seamlessly integrated

into functions.

- **Creating Azure Functions:**
 - You can create Azure Functions using the Azure Portal, Visual Studio, Azure CLI, or other development tools.
 - Select an appropriate trigger type based on the event that should trigger your function's execution.
- **Function Configuration:**
 - Configure input and output bindings within your function to interact with various Azure services like Blob Storage, Event Hub, and more.
- **Testing and Debugging:**
 - Test your functions locally using tools like Azure Functions Core Tools.
 - Debug functions with built-in debugging capabilities.
- **Deployment and Scaling:**
 - Deploy your Azure Functions to the Azure cloud for production use.
 - Azure Functions auto-scale based on demand, ensuring resource efficiency.
- **Use cases:**
 - Azure Functions are suitable for a wide range of use cases, including data processing, webhooks, IoT event handling, and more.

Exercises

1. Reproduce the case-study.
2. Explore other input bindings.
3. Explorer other trigger bindings.
4. Explore other output bindings.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 5

Azure SQL, Cosmos DB and Blob Storage

Introduction

In today's digital landscape, the demand for efficient and scalable data storage and management solutions has grown exponentially. Microsoft Azure offers a comprehensive suite of cloud-based services that cater to diverse data storage and management requirements. This chapter will delve into three key components of Azure's data services: Azure SQL, Cosmos DB, and Blob Storage.

Azure SQL is a fully managed relational database service provided by Azure. It enables you to build, deploy, and scale relational applications with ease. Whether you are working on a small-scale project or managing large enterprise-level applications, Azure SQL provides a reliable and secure platform for your data storage needs. This chapter will provide an overview of Azure SQL, discussing its features, benefits, and how it fits into the broader Azure ecosystem. Additionally, we will explore various usage examples that showcase the versatility and applicability of Azure SQL in real-world scenarios.

Cosmos DB, on the other hand, is a globally distributed, multi-

model database service designed to handle massive-scale applications seamlessly. With Cosmos DB, you can store and retrieve data using a variety of models, including document, key-value, column-family, graph, and more. Its impressive scalability, low-latency performance, and global distribution capabilities make Cosmos DB an ideal choice for building highly responsive and globally accessible applications. Throughout this chapter, we will provide an overview of Cosmos DB, highlighting its key features and discussing practical usage examples that illustrate its potential in different domains.

Blob Storage, the third component covered in this chapter, is Azure's storage solution specifically designed for storing unstructured data such as images, videos, documents, and logs. Blob Storage offers scalable and cost-effective storage options, allowing you to easily manage and access your unstructured data in the cloud. In this chapter, we will explore the various features of Blob Storage and discuss how it can be leveraged in different scenarios. Additionally, we will provide practical usage examples that demonstrate the versatility and benefits of Blob Storage for handling unstructured data.

By the end of this chapter, you will have a solid understanding of Azure SQL, Cosmos DB, and Blob Storage. You will be equipped with the knowledge needed to leverage these services effectively, enabling you to build robust and scalable solutions that meet your data storage and management requirements in the Azure cloud environment. So, let us dive in and explore the world of Azure SQL, Cosmos DB, and Blob Storage!

Structure

This chapter covers the following topics:

- Azure SQL, Cosmo DM and Blob Storage
- Azure SQL
- Cosmos DB
- Blob Storage

Objectives

Gain a comprehensive understanding of key Azure services by chapter's end, delving into Azure SQL's fundamentals and its pivotal role in the data services ecosystem. Explore its features, emphasizing scalability and security, with practical insights from various usage examples. Transition seamlessly to Cosmos DB, a globally distributed, multi-model database service, understanding diverse data models and real-world performance attributes. Focus on Blob Storage as Azure's solution for unstructured data, covering scalability and cost-effectiveness. Acquire skills for effective utilization of Azure SQL, Cosmos DB, and Blob Storage, empowering readers to architect and manage solutions within the Azure landscape.

Azure SQL, Cosmos DB and Blob Storage

This chapter provides an overview of the three key Azure storage services and explains how to use them in .NET applications. The chapter covers Azure SQL Database, Azure Cosmos DB, and Azure Blob Storage, and provides practical examples for each service.

Azure SQL

Azure SQL is a fully managed relational database service provided by Microsoft Azure. It offers a reliable and scalable platform for storing and managing relational data in the cloud. Azure SQL eliminates the need for managing infrastructure and provides a range of features and capabilities that enable developers to focus on building applications rather than managing database servers.

This service is built on the Microsoft SQL Server engine and supports a wide range of familiar tools and technologies used in SQL Server environments. Azure SQL offers compatibility with existing SQL Server applications, making it seamless for organizations to migrate their on-premises databases to the cloud.

Key features of Azure SQL:

- **Scalability:** Azure SQL can scale up or down based on demand, allowing applications to handle varying workloads efficiently.
- **High availability:** It provides built-in high availability with automatic failover and redundancy, ensuring continuous

access to data.

- **Security:** Azure SQL offers robust security features, including data encryption, threat detection, and identity management, to protect sensitive information.
- **Managed service:** Microsoft manages the underlying infrastructure, taking care of tasks like patching, backups, and maintenance, so users can focus on application development.
- **Integration:** Azure SQL integrates seamlessly with other Azure services, allowing for data integration, analytics, and application development within the Azure ecosystem.

Usage examples of Azure SQL:

- **Web applications:** Azure SQL is commonly used as the backend database for web applications, providing a scalable and reliable data storage solution.
- **Line-of-business applications:** It is suitable for building and managing business-critical applications that require secure and highly available data storage.
- **Data warehousing:** Azure SQL can be utilized for data warehousing, enabling organizations to analyze and process large volumes of structured data.
- **Reporting and analytics:** It supports reporting and analytics solutions, allowing users to gain insights from data and generate meaningful reports.
- **Software as a Service (SaaS):** Azure SQL is commonly used by SaaS providers as the database backend for their multi-tenant applications.

By leveraging Azure SQL, organizations can benefit from the advantages of a fully managed, scalable, and secure relational database service in the Azure cloud environment. It provides the necessary tools and features to build and manage high-performance applications without the overhead of infrastructure management, allowing businesses to focus on innovation and growth.

Scaling Azure SQL Server

Scaling Azure SQL Server involves adjusting the resources allocated to the server to meet the demands of your applications and workloads. Azure SQL Server provides several scaling options to

accommodate varying performance and capacity requirements. Here are the key approaches to scaling Azure SQL Server:

- Scaling compute
 - Azure SQL provides two compute options: Provisioned and serverless.
 - **Provisioned compute:** In this mode, you choose a specific amount of compute resources (CPU and memory) for your SQL server. You can scale up or down the compute resources as needed, but it may involve downtime during the scaling operation.
 - **Serverless compute:** This mode automatically scales compute resources based on workload demands. It allows for automatic pausing and resuming of the server during periods of inactivity to save costs.
- Scaling storage
 - Azure SQL Server supports automatic storage scaling. Storage capacity is adjusted automatically based on your database's size and usage. You don't need to worry about managing storage manually.
- Elastic Pool
 - Azure SQL Elastic Pool is a resource allocation model that allows you to share resources across multiple databases within a pool.
 - By grouping databases with similar resource requirements into a pool, you can optimize resource utilization and cost efficiency.
 - Elastic Pool provides automatic scaling of resources to meet the collective demands of the databases within the pool.
- Hyperscale
 - Azure SQL Hyperscale is designed for large-scale and highly demanding workloads.
 - It offers nearly limitless storage capacity and supports scaling compute resources dynamically without any downtime.
 - Hyperscale enables seamless scaling of both storage and compute to handle data-intensive workloads effectively.
- Geo-replication

- Azure SQL Server supports geo-replication, allowing you to create read-only replicas of your databases in different Azure regions.
- Replication provides high availability and enables scaling by distributing read workloads across multiple regions.
- Managed instance
 - Azure SQL Managed Instance is a fully managed SQL Server instance in the cloud.
 - Scaling in managed instance involves choosing the appropriate service tier and specifying the required compute and storage resources.

When scaling Azure SQL Server, consider factors such as performance requirements, workload patterns, and cost optimization. Regular monitoring of resource utilization and workload patterns helps determine when and how to scale the server effectively. Azure Portal, Azure PowerShell, Azure CLI, or automation tools like Azure Functions can be used to manage and automate scaling operations.

Usage example

In this practical example, we will walk through the process of creating a web app and establishing a connection with an Azure SQL Server. By following these steps, you will be able to set up a web application that securely interacts with the Azure SQL Server for data storage and retrieval.

Creating the Azure resource

To create an Azure SQL Server, follow these steps:

1. Sign in to the Azure portal (portal.azure.com) using your Azure account credentials.
2. In the left-hand navigation pane, click on **"Create a resource"** and search for **"SQL Server"** in the search bar.
3. Select **"SQL Server"** from the list of available resources.
4. In the **"SQL Server"** blade, click on the **"Create"** button to begin the creation process.
5. Provide a unique server name and choose the subscription, resource group, and region where you want to create the

SQL Server instance.

6. Select the desired version and edition of SQL Server to be deployed.
7. Choose the authentication method for accessing the server. You can either use a system-assigned username and password or Azure Active Directory authentication.
8. Configure the networking settings, including firewall rules, to allow access to the SQL Server from specific IP addresses or Azure services.
9. Optionally, enable advanced data security features such as Threat Detection and Vulnerability Assessment for enhanced security.
10. Review the configuration settings and click on "**Review + Create**" to validate your choices.
11. After validation, click on "**Create**" to begin the deployment process. Azure will create the SQL Server instance based on your specifications.
12. Once the deployment is complete, you can access the SQL Server using tools such as Azure Data Studio and **SQL Server Management Studio (SSMS)**, or programmatically using various client libraries and APIs.

Creating an Azure SQL Server provides you with a fully managed and scalable relational database service in the Azure cloud. It allows you to store and manage your data efficiently, ensuring high availability, security, and performance for your applications. In the following figure we have an overview of a SQL Server.

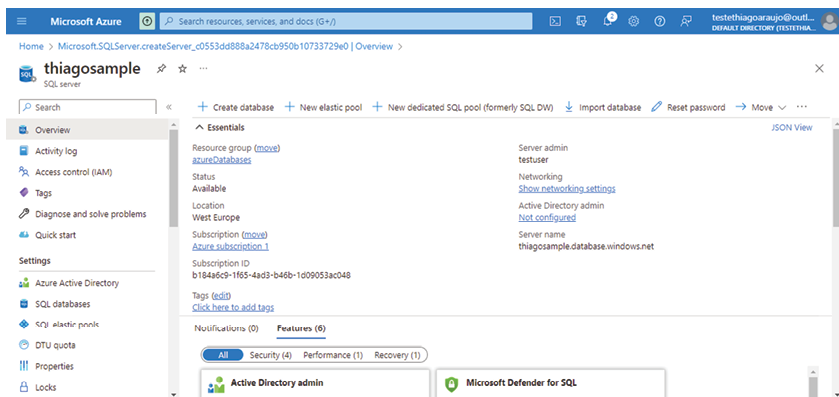


Figure 5.1: Azure SQL Server Overview

After successfully creating the SQL Server, the next step is to create a database within the server. The database serves as the container for organizing and storing your data. Follow these steps to create a database in Azure SQL Server:

1. Access the Azure portal (portal.azure.com) using your Azure account credentials.
2. Navigate to the Azure SQL Server section and select the specific server where you want to create the database.
3. Within the server's blade, locate the **"Databases"** section and click on the **"Create database"** button.
4. Provide a unique name for the database, keeping in mind any naming conventions or project-specific requirements.
5. Specify the desired database settings, such as the collation, which determines the sorting and comparison rules for character data.
6. Choose the appropriate pricing tier based on your performance and capacity needs. Azure offers different tiers, including Basic, Standard, and Premium, each with varying performance and features.
7. Configure additional settings, such as enabling Advanced Threat Protection or enabling Geo-Redundant Backup for enhanced security and data protection.
8. Review the configuration details and click on the **"Review + Create"** button to validate your choices.

9. After validation, click on "**Create**" to initiate the database creation process. Azure will provision the database within the specified SQL Server.

In the following figure, we have an overview of A SQL Server Database.

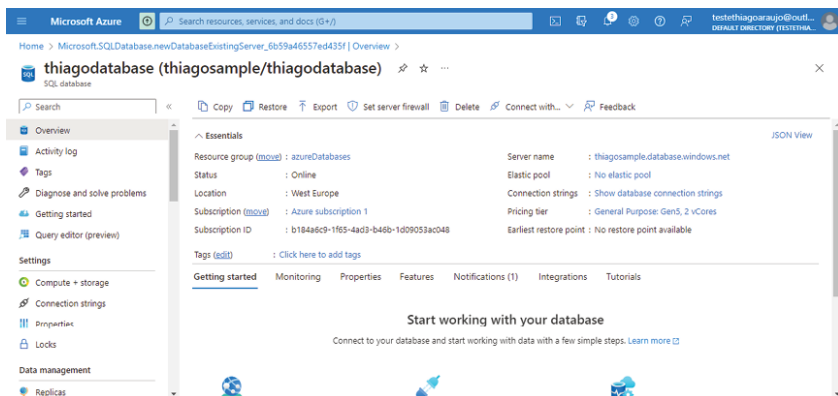


Figure 5.2: SQL Database Overview

Connecting to the database

As we embark on creating the web application, let us begin by enriching our project with essential NuGet packages. These packages bring a wealth of functionality and libraries that will enhance our development process and empower our application. To get started, we recommend including latest versions of the following NuGet packages:

- Microsoft.EntityFrameworkCore
- Microsoft.EntityFrameworkCore.SqlServer
- Microsoft.EntityFrameworkCore.Tools

Add two classes, one as our **dbContext** and one as our entity:

The **DbContext** class:

1.

```
public class SqlServerDbContext :  
    DbContext
```
2.

```
{
```
3.

```
    public SqlServerDbContext (
```
- 4.

```

DbContextOptions<SqlServerDbContext>
options)
5.         : base(options)
6.         {
7.         }
8.
9.         public DbSet<SampleEntity>
SampleEntities { get; set; }
10.    }

```

The entity class:

```

1. public class SampleEntity
2.     {
3.         public int Id { get; set; }
4.         public string Description { get;
set; }
5.         public DateTime CreationDate { get;
set; }
6.     }

```

Getting the connection string, this connection string is authenticating with Azure Directory but we have chosen to use Sql Server. You might see a different connection string in the project due to this choice. In the following figure you can see the connection string section of the recently created SQL Server Database:

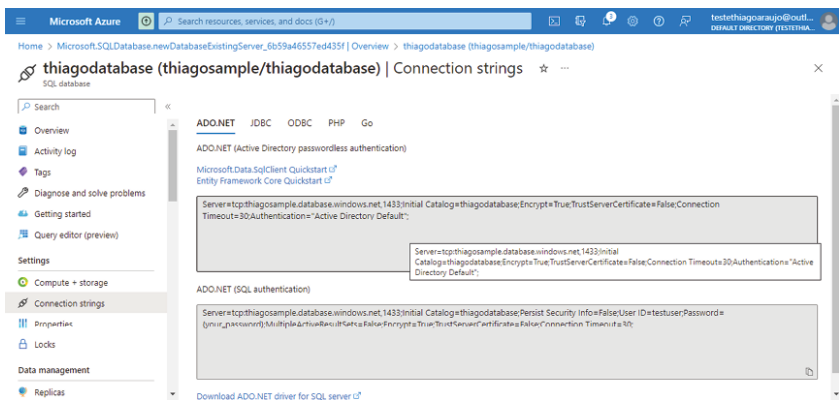


Figure 5.3: SQL Database Connection Strings.

When setting up an Azure SQL Database, it is essential to configure the database firewall to whitelist your IP address to ensure secure and controlled access to your database. By default, Azure SQL Database is protected by a firewall that blocks all external connections to secure your data from unauthorized access. You might need to configure the SQL Database firewall to whitelist your IP address, this ensure that your application from the specified IP can connect to the Azure SQL Database securely, as follows:

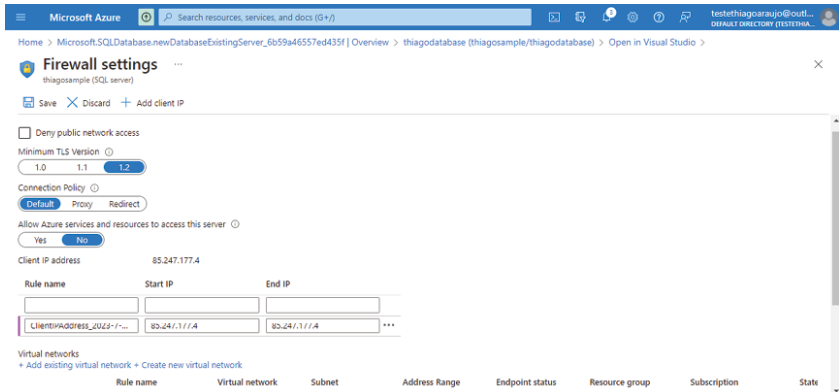


Figure 5.4: SQL Server Firewall settings.

If the firewall settings are not configured, then the following error will prompt:

Error 40615

Message text: "Cannot open server '{0}' requested by the login. Client with IP address '{1}' is not allowed to access the server."

Updating our **program.cs** with the connection string:

1. `using Microsoft.EntityFrameworkCore;`
2. `using SqlServerWebApp.Database;`
- 3.
4. `var builder =`
`WebApplication.CreateBuilder(args);`
- 5.
6. *// Add services to the container.*
7. `builder.Services.AddRazorPages();`
8. `builder.Services.AddDbContext<SqlServerDbContext>`

```
9.         options => options.UseSqlServer(
10. "Data
    Source=thiagosample.database.windows.net; "+
11. "Initial Catalog=thiogodatabase;User
    ID=testuser; "+
12. "Password=Testpassword##;Connect
    Timeout=60; "+
13. "Encrypt=True;Trust Server
    Certificate=False; "+
14. "Application Intent=ReadWrite;Multi Subnet
    Failover=False»));
15.
16.
17.
18. var app = builder.Build();
19.
20. // Configure the HTTP request pipeline.
21. if (!app.Environment.IsDevelopment())
22. {
23.     app.UseExceptionHandler("/Error");
24.     // The default HSTS value is 30 days. You may want to
    change this for production scenarios, see https://aka.ms/
    aspnetcore-hsts.
25.     app.UseHsts();
26. }
27.
28. app.UseHttpsRedirection();
29. app.UseStaticFiles();
30.
31. app.UseRouting();
32.
33. app.UseAuthorization();
34.
35. app.MapRazorPages();
36.
37. app.Run();
```

Running migration commands:

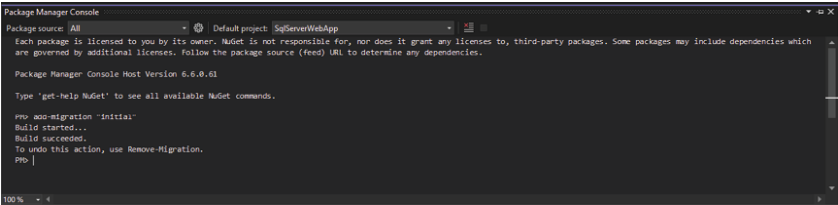


Figure 5.5: Package Manager Consoler with add-migration command.

Update database:

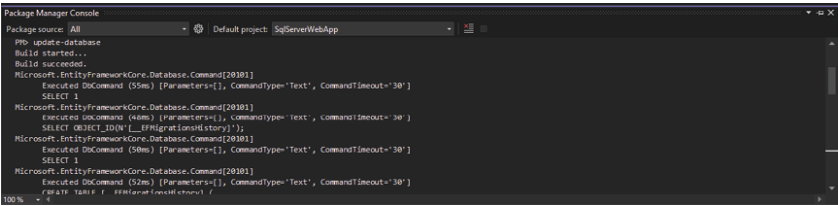


Figure 5.6: Package Manager Consoler with update-database command.

Validating database creation through Azure Data Studio:

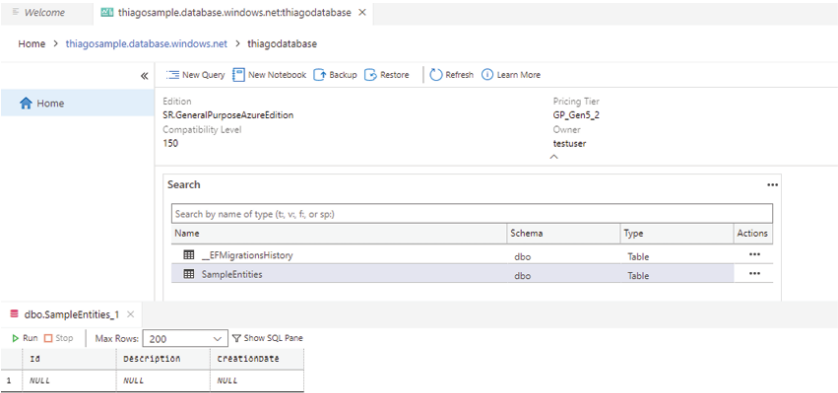


Figure 5.7: SQL Database with recently created data.

Cosmos DB

Cosmos DB is a globally distributed, multi-model database service provided by Microsoft Azure. It is designed to handle massive-scale

applications with ease, providing high throughput, low-latency performance, and global reach. Cosmos DB supports various data models, including document, key-value, column-family, graph, and more, making it a versatile choice for diverse application requirements.

Key features of Cosmos DB:

- **Global distribution:** Cosmos DB enables data to be replicated across multiple regions worldwide, ensuring low-latency access and high availability for users across the globe.
- **Scalability:** It offers horizontal scalability with automatic partitioning, allowing applications to scale seamlessly as data volumes and throughput requirements grow.
- **Multi-model support:** Cosmos DB supports multiple data models, allowing developers to choose the most suitable model for their applications and switch between them seamlessly.
- **Consistency levels:** It provides a range of consistency options, allowing developers to choose the level of consistency required for their applications, from strong to eventual consistency.
- **Turnkey global distribution:** Cosmos DB takes care of data replication and distribution across regions, abstracting the complexity and allowing developers to focus on building applications.

Usage examples of Cosmos DB:

- **High-volume web and mobile applications:** Cosmos DB is ideal for applications that require low-latency access to data across the globe, such as e-commerce platforms or social media networks.
- **IoT and telemetry data:** It can handle massive streams of IoT and telemetry data, storing and processing high-volume data points in real-time.
- **Personalized user experiences:** Cosmos DB supports graph databases, making it suitable for applications that require complex relationship mapping, such as recommendation engines or social networks.
- **Time-series data:** It can efficiently store and process time-series data, making it suitable for applications in finance, IoT,

and monitoring systems.

- **Content management and catalogs:** cosmos DB can be used to manage and serve content catalogs, allowing efficient querying and retrieval of large amounts of structured data.

Cosmos DB Containers

Azure Cosmos Containers provide scalability for both storage and throughput in Azure Cosmos DB. They are also advantageous when you require different configurations among your Azure Cosmos DBs, as each container can be individually configured.

Azure Cosmos Containers have specific properties, which can be system-generated or user-configurable, depending on the API used. These properties range from unique identifiers for containers to purging policy configurations. You can find the complete list of properties for each API in the documentation.

During creation, you can choose between two throughput strategies:

- **Dedicated mode:** In this mode, the provisioned throughput is exclusively allocated to the container and comes with SLAs.
- **Shared mode:** In this mode, the provisioned throughput is shared among all containers operating in shared mode.

Cosmos DB Containers are available for all Cosmos DB APIs except the Gremlin API and Table API.

Scaling Cosmos DB

Azure Cosmos DB offers manual and automatic scaling options without service interruption or impact on Azure Cosmos DB SLAs.

With automatic scaling, Azure Cosmos DB dynamically adjusts your throughput capacity based on usage without the need for manual logic or code. You simply set the maximum throughput capacity, and Azure adjusts the throughput within the range of 10% to 100% of the maximum capacity.

Manual scaling allows you to change the throughput capacity permanently according to your requirements.

It is crucial to choose partition keys wisely before scaling Azure Cosmos DB to avoid hot partitions, which can increase costs and degrade performance.

When configuring autoscale, consider important topics such as:

- **Time to Live (TTL):** Define the TTL for the container, enabling item-specific or all-items expiration.
- **Geospatial Configuration:** Query items based on location.
- **Geography:** Represents data in a round-earth coordinate system.
- **Geometry:** Represents data in a flat coordinate system.
- **Partition Key:** The key used for partitioning and scaling.
- **Indexing Policy:** Define how indexes are applied to container items, including properties to include or exclude, consistency modes, and automatic index application.

Triggers, stored procedures, and UDFs

Azure Cosmos DB allows the execution of code through Triggers, Stored Procedures, and Functions. These can be defined using the Azure Portal, JavaScript Query API for Cosmos DB, or Cosmos DB SQL API client SDKs and you can find a better explanation of them below:

- **Triggers:**
 - Triggers in Cosmos DB are event-driven and can be defined on collections to execute custom logic before or after specific operations on the data.
 - There are two types of triggers: pre-triggers and post-triggers.
- Pre-triggers are executed before a data operation (insert, update, delete) occurs, allowing you to validate or modify the incoming data.
- Post-triggers are executed after a data operation, providing an opportunity to perform additional actions or trigger external processes.
- Triggers can be written in JavaScript and are executed within the Cosmos DB environment.
- **Stored procedures:**
 - Stored procedures are JavaScript functions stored and executed within Cosmos DB.
 - They enable the execution of complex server-side logic,

including conditional branching, looping, and multiple database operations, in an atomic manner.

- Stored procedures can be used to perform batch updates, transactional operations, data aggregation, and more.
- They are often used for performance optimizations by reducing network round trips and executing multiple operations as a single unit of work.
- **User-Defined Functions (UDFs):**
 - UDFs are JavaScript functions that allow you to define custom logic that can be called within queries and stored procedures.
 - UDFs are similar to functions in traditional programming languages and can encapsulate reusable code for data transformation or calculations.
 - They can be used within SQL queries to perform custom computations or transformations on data retrieved from Cosmos DB collections.
 - UDFs can greatly enhance the flexibility and expressiveness of queries by encapsulating complex calculations or data transformations.

Triggers, stored procedures, and UDFs provide a means to extend the functionality of Cosmos DB beyond basic CRUD operations. They allow for complex data manipulation, validation, and custom logic execution within the database itself, reducing the need for round trips to external systems. These features can help improve performance, reduce network traffic, and simplify application development by pushing processing logic closer to the data.

Change feed notifications with Cosmos DB

Azure Cosmos DB change feed notifications service monitors changes across containers and distributes events triggered by those changes to multiple consumers.

The main components of change feed notifications include:

- **Monitored container:** The container where inserts or updates trigger operations are reflected in the change feed.
- **Lease container:** Stores states and coordinates the change feed processor.

- **Host:** The environment hosting the change feed processor.
- **Delegate:** Code executed when events are triggered by the change feed notifications.

The change feed processor can be hosted on various Azure services that support long-running tasks, such as Azure WebJobs, Azure Virtual Machines, Azure Kubernetes Services, and Azure .NET hosted services.

By leveraging the capabilities of Cosmos DB, organizations can build globally distributed, highly responsive, and scalable applications. Its versatility in supporting multiple data models and seamless integration with other Azure services makes it a powerful tool for modern application development. With Cosmos DB, developers can focus on building innovative solutions while benefiting from the global reach and performance optimizations provided by the service.

Usage examples of Cosmos DB for NoSQL

In this practical example, we will guide you through the process of creating a web application and establishing a seamless connection with Azure Cosmos DB. By following these steps, you will be able to set up a robust web app that leverages the power of Azure Cosmos DB for efficient data storage and retrieval.

Creating the Azure Resource

To create an Azure Cosmos DB account for a NoSQL database, follow these steps:

1. Sign in to the Azure portal (portal.azure.com) using your Azure account credentials.
2. Click on the "**Create a resource**" button (+) on the top-left corner of the portal.
3. Search for "**Azure Cosmos DB**" in the search bar or browse through the available services.
4. Select "**Azure Cosmos DB**" from the search results.
5. In the "**Azure Cosmos DB**" blade, click on the "**Create**" button to begin the account creation process.
6. Provide the necessary details for the Cosmos DB account:

- a. Choose the appropriate subscription.
 - b. Create a new resource group or select an existing one to contain the account.
 - c. Specify a unique account name for your Cosmos DB account.
 - d. Select the desired API (e.g., Core SQL for document data, MongoDB, Cassandra, Gremlin, or Table) based on your NoSQL database requirements.
 - e. Choose the preferred location or region where you want to deploy the Cosmos DB account.
 - f. Enable multi-region writes for globally distributed data if necessary.
 - g. Configure the desired consistency level based on your application's needs.
7. Review the configuration settings, including the pricing tier and additional features.
 8. Click on the **"Review + Create"** button to validate your choices.
 9. After validation, click on **"Create"** to initiate the provisioning process. Azure will create the Cosmos DB account according to your specifications.

In the following figure, you can see how to create an Azure Cosmos DB for NoSQL:

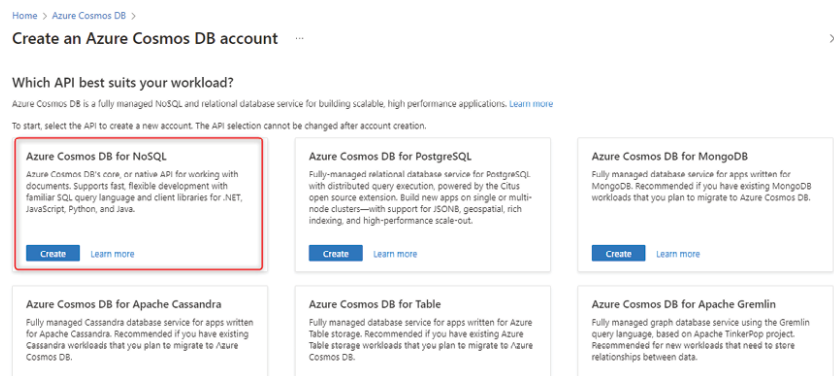


Figure 5.8: Azure Cosmos DB for NoSQL account creation.

This is the output when you have your Azure Cosmos DB account created:

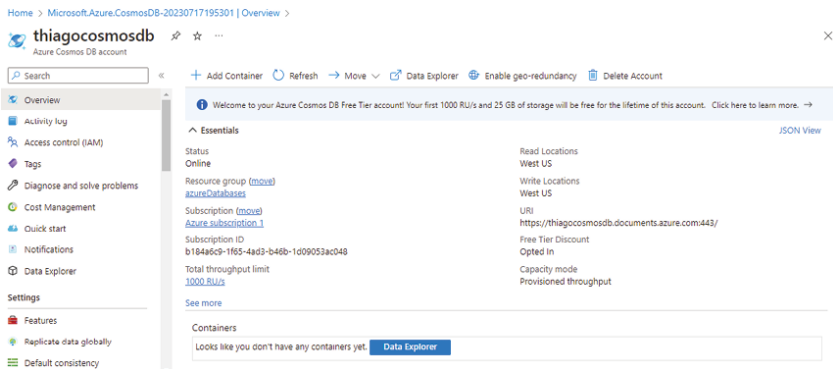


Figure 5.9: Azure Cosmos DB Overview.

Connecting to the Database

Once the Cosmos DB account is successfully created, you can access it through the Azure portal.

From the Cosmos DB account blade, you can manage your NoSQL database, configure data replication, and access connection settings such as the primary connection string.

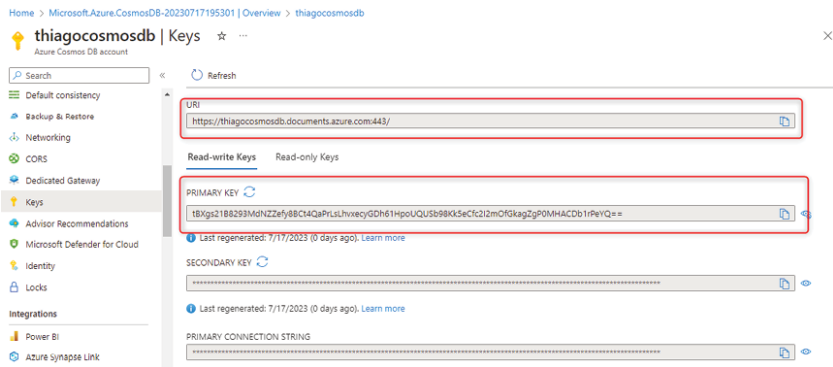


Figure 5.10: Azure Cosmos DB Keys.

Now, in our web application, we must install the latest version of following Nuget package:

Microsoft.Azure.Cosmos

Now we have created three classes:

```
1.     public class Address
2.     {
3.         [JsonProperty(PropertyName = "id")]
4.         public string Id { get; set; }
5.         public string City { get; set; }
6.         public string StreetAndNumber {
7.             get; set; }
8.     }
9.     public class Person
10.    {
11.        [JsonProperty(PropertyName = "id")]
12.        public string Id { get; set; }
13.        public DateTime BirthDate { get;
14.            set; }
15.        public string Name { get; set; }
16.        public string LastName { get; set;
17.    }
18.        public Address Address { get; set;
19.    }
20.        public Vehicle Vehicle { get; set;
21.    }
22.    }
23.    public class Vehicle
24.    {
25.        [JsonProperty(PropertyName = "id")]
26.        public string Id { get; set; }
27.        public int Year { get; set; }
28.        public string Model { get; set; }
29.        public string Make { get; set; }
30.    }
```

We have also created a helper to populate the DB with some data:

```
1.     public static class CreatePerson
2.     {
3.         public static Person GetNewPerson()
4.         {
```

```

5.         Random random = new Random();
6.         return new Person
7.         {
8.             BirthDate =
DateTime.Now.AddYears(28),
9.             Id = random.Next() +
"Thiago",
10.            Name = "Thiago",
11.            LastName = "Araujo",
12.            Vehicle = new Vehicle
13.            {
14.                Id = random.Next() +
"BMW",
15.                Make = "BMW",
16.                Model = "116D",
17.                Year = random.Next()
18.            },
19.            Address = new Address
20.            {
21.                Id = random.Next() +
"Portugal",
22.                City = "Lisbonne",
23.                StreetAndNumber =
"Avenida da Liberdade, 25"
24.            }
25.        };
26.    }
27. }

```

We had to update the **program.cs** to inject the CosmosDB client:

```

1. using Microsoft.Azure.Cosmos;
2.
3. var builder =
WebApplication.CreateBuilder(args);
4.
5. // Add services to the container.
6. builder.Services.AddRazorPages();

```



```

7.
8. SocketsHttpHandler socketsHttpHandler = new
   SocketsHttpHandler();
9. // Customize this value based on desired DNS refresh timer
10. socketsHttpHandler.PooledConnectionLifetime
    = TimeSpan.FromMinutes(5);
11. // Registering the Singleton SocketsHttpHandler lets you reuse
    it across any HttpClient in your application
12. builder.Services.AddSingleton<SocketsHttpHandler>
13.
14. // Use a Singleton instance of the CosmosClient
15. builder.Services.AddSingleton<CosmosClient>(serv
    =>
16. {
17.     SocketsHttpHandler socketsHttpHandler =
        serviceProvider
18.
19.     .GetRequiredService<SocketsHttpHandler>();
20.     CosmosClientOptions cosmosClientOptions
    =
21.         new CosmosClientOptions()
22.         {
23.             HttpClientFactory = () => new
                HttpClient(socketsHttpHandler,
                    disposeHandler: false)
24.         };
25.     return new CosmosClient(
26.         "https://
            thiagocosmosdb.documents.azure.com:443/",
27.         "tBXgs21B8293MdNZZefy8BCt4QaPrLsLhvxecyGDh61HpoU
            2mOfGkagZgP0MHACDb1rPeYQ==",
28.         cosmosClientOptions);
29. });
30.
31.
32. var app = builder.Build();
33.

```

```

34. // Configure the HTTP request pipeline.
35. if (!app.Environment.IsDevelopment())
36. {
37.     app.UseExceptionHandler("/Error");
38.     // The default HSTS value is 30 days. You may want to
       change this for production scenarios, see https://aka.ms/
       aspnetcore-hsts.
39.     app.UseHsts();
40. }
41.
42. app.UseHttpsRedirection();
43. app.UseStaticFiles();
44.
45. app.UseRouting();
46.
47. app.UseAuthorization();
48.
49. app.MapRazorPages();
50.
51. app.Run();
52.

```

And this is the implementation of our CosmosDB for NOSql in the **index.cshtml.cs** class:

```

1. using CosmosDBWebApp.Helper;
2. using CosmosDBWebApp.Models;
3. using Microsoft.AspNetCore.Mvc;
4. using Microsoft.AspNetCore.Mvc.RazorPages;
5. using Microsoft.Azure.Cosmos;
6.
7. namespace CosmosDBWebApp.Pages
8. {
9.     public class IndexModel : PageModel
10.    {
11.        private readonly
       ILogger<IndexModel> _logger;

```

```

12.         private readonly CosmosClient
           cosmosClient;
13.         private readonly string
           databaseName = "ThiagoCosmosDB";
14.         private readonly string
           sourceContainerName =
15.             "ThiagoContainer";
16.
17.         public
           IndexModel(ILogger<IndexModel> logger,
18.                   CosmosClient
           cosmosClient)
19.         {
20.             this._logger = logger;
21.             this.cosmosClient =
           cosmosClient;
22.         }
23.
24.         public async Task OnGet()
25.         {
26.             DatabaseResponse
           databaseResponse =
27.                 await cosmosClient
28.
           .CreateDatabaseIfNotExistsAsync(databaseName);
29.             Database database =
           databaseResponse.Database;
30.
31.             ContainerResponse container =
32.                 await
           database.CreateContainerIfNotExistsAsync(
33.                 new
           ContainerProperties(sourceContainerName, "/"
           id"));
34.
35.             await
           CreateItemsAsync(cosmosClient,

```

```

36.             database.Id,
           container.Container.Id);
37.         }
38.         private static async Task
CreateItemsAsync(
39.             CosmosClient cosmosClient,
           string databaseId,
40.             string containerId)
41.         {
42.             Container sampleContainer =
cosmosClient
43.
           .GetContainer(databaseId, containerId);
44.
45.             for (int i = 0; i < 15; i++)
46.             {
47.                 var person =
CreatePerson.GetNewPerson();
48.                 await sampleContainer
49.
           .CreateItemAsync<Person>(person,
50.
           new
           PartitionKey(person.Id));
51.             }
52.         }
53.     }
54. }

```

This is the output, with our container created and data in it:

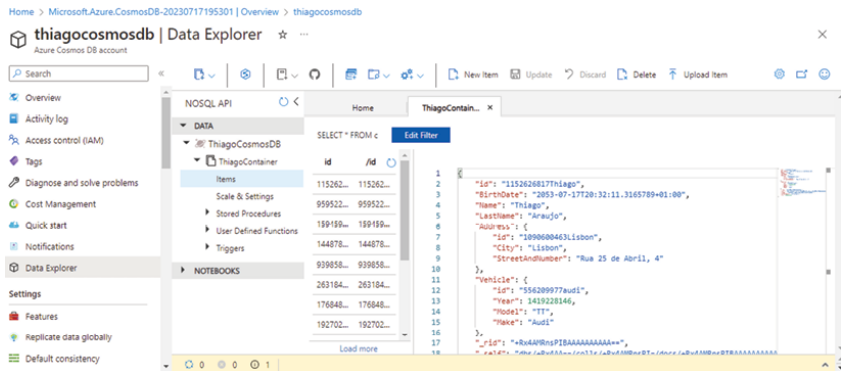


Figure 5.11: Azure Cosmos DB Data Explorer with recently created data.

Blob Storage

Blob Storage is a storage solution provided by Microsoft Azure that is designed specifically for storing unstructured data, such as images, videos, documents, logs, and other large files. It offers scalable and cost-effective storage options, making it easy to manage and access unstructured data in the cloud.

Following are some key features of Blob Storage:

- **Scalability:** Blob Storage can scale to accommodate the storage needs of any application, whether it is a small-scale project or a high-traffic enterprise application.
- **Cost-effectiveness:** It provides flexible pricing options, allowing users to choose storage tiers based on their data access frequency and cost requirements.
- **Durability and availability:** Blob Storage ensures durability by storing multiple copies of data across different storage nodes. It also offers built-in redundancy and high availability for data access.
- **Easy accessibility:** Blob Storage provides multiple access methods, allowing users to easily retrieve and manage their data programmatically or through Azure portal, APIs, or command-line tools.
- **Integration:** It seamlessly integrates with other Azure services, enabling users to leverage Blob Storage as a backend for various applications, data processing, and analytics workflows.

Usage examples of Blob Storage:

- **Media storage and delivery:** Blob Storage is commonly used to store and deliver media files, such as images, videos, and audio files, for web applications or content management systems.
- **Backup and disaster recovery:** It provides an efficient and cost-effective solution for backing up data and creating disaster recovery copies in the cloud.
- **Logging and analytics:** Blob Storage can be used to store logs generated by applications or systems, enabling later analysis and processing for operational insights.
- **Archiving and compliance:** It is suitable for long-term archival of data that needs to be stored for regulatory compliance or historical purposes.
- **Data Distribution:** Blob Storage supports the distribution of large datasets across multiple regions, allowing efficient global access to data by users.

By leveraging Blob Storage, organizations can effectively manage and store unstructured data in the cloud. Its scalability, cost-effectiveness, and easy accessibility make it a valuable tool for a wide range of applications, from media storage and delivery to backup and analytics. Whether it is a small-scale project or a large enterprise solution, Blob Storage provides a reliable and efficient way to manage unstructured data in the Azure cloud environment.

Scaling Blob Storage

Scaling Blob Storage in Azure involves adjusting the storage capacity and performance to meet your data storage and retrieval needs. Blob Storage provides flexible scaling options to accommodate varying workloads and storage requirements. Here are the key approaches to scaling Blob Storage:

- **Capacity scaling**
 - Blob Storage allows you to scale the storage capacity by increasing or decreasing the amount of data you can store.
 - You can easily add more storage by provisioning additional storage accounts or expanding the existing storage accounts to handle larger data volumes.

- Azure provides virtually limitless storage scalability, allowing you to scale up as your storage needs grow.
- **Performance scaling**
 - Blob Storage provides two performance tiers: Hot and cool.
 - Hot access tier offers higher performance and is optimized for frequently accessed data.
 - Cool access tier provides lower-cost storage for data that is less frequently accessed.
 - You can choose the appropriate tier based on the access patterns and frequency of data retrieval to optimize costs and performance.
- **Parallelism and throughput scaling**
 - Azure Blob Storage can handle high-throughput scenarios by parallelizing access to the storage accounts.
 - By distributing the workload across multiple storage accounts, you can achieve higher throughput and reduce latency.
 - You can leverage techniques such as sharding, parallel processing, and load balancing to optimize performance and scale horizontally.
- **Content Delivery Network (CDN)**
 - Azure CDN can be integrated with Blob Storage to scale content delivery globally.
 - CDN caches and delivers content from Blob Storage to users around the world, reducing latency and improving performance.
 - By leveraging Azure CDN, you can scale the delivery of static content, such as images, videos, and documents, to a global audience.
- **Lifecycle management policies**
 - Blob Storage provides lifecycle management policies to automate the movement and deletion of data based on defined rules.
 - By configuring lifecycle management policies, you can automatically move less frequently accessed data to the Cool Access Tier or archive it to Azure Archive Storage to

optimize costs.

When scaling Blob Storage, consider factors such as data growth rate, access patterns, performance requirements, and cost optimization. Regular monitoring of storage utilization and access patterns helps determine when and how to scale the storage effectively. Azure Portal, Azure PowerShell, Azure CLI, or automation tools can be used to manage and automate scaling operations.

Usage example

In this practical example, we will guide you through the process of creating a web application and establishing a seamless connection with Azure Blob Storage. By following these steps, you will be able to set up a robust web app that leverages the power of Azure Blob Storage for efficient storage and retrieval of unstructured data.

Creating the Azure Resource

To create an Azure Storage account, follow these steps:

1. Sign in to the Azure portal (portal.azure.com) using your Azure account credentials.
2. Click on the "**Create a resource**" button (+) on the top-left corner of the portal.
3. Search for "**Storage account**" in the search bar or browse through the available services.
4. Select "**Storage account**" from the search results.
5. In the "**Storage account**" blade, click on the "**Create**" button to begin the account creation process.
6. Provide the necessary details for the storage account:
 - a. Choose the appropriate subscription.
 - b. Create a new resource group or select an existing one to contain the storage account.
 - c. Specify a unique name for your storage account.
 - d. Select the desired location or region where you want to deploy the storage account.
 - e. Choose the desired performance tier: Standard (for general-purpose storage) or Premium (for high-

performance SSD storage).

- f. Select the desired account kind: Blob storage, General-purpose v1, General-purpose v2, or BlockBlobStorage (optimized for block blob storage scenarios).
 - g. Configure additional settings such as replication options, access tiers, and encryption.
7. Review the configuration settings, including the pricing tier and additional features.
 8. Click on the **"Review + Create"** button to validate your choices.
 9. After validation, click on **"Create"** to initiate the provisioning process. Azure will create the storage account according to your specifications.

In the following figure, we can see an overview of the Storage Account recently created:

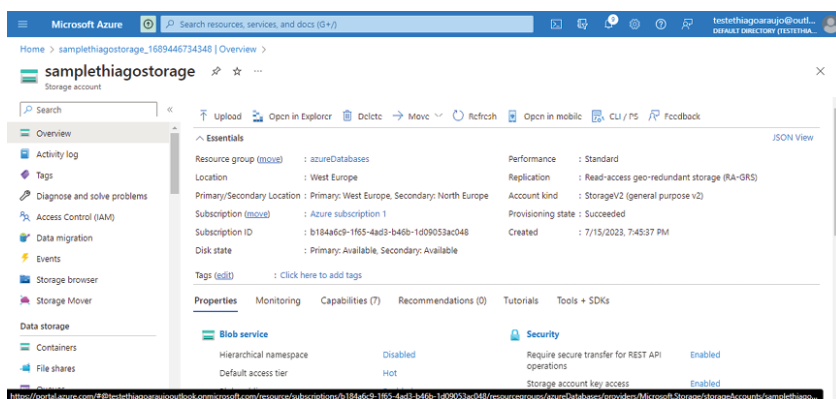


Figure 5.12: Azure Storage Account Overview.

1. Within the Storage account blade, locate the **"Blob service"** or **"Containers"** section and click on the **"+ Container"** button.
2. In the **"Create container"** blade, provide a unique name for the container. The name must be lowercase and can include letters, numbers, and hyphens.
3. Choose the desired level of public access for the container:
 - **Private:** Only accessible with the appropriate access keys

or shared access signatures.

- **Blob:** Allows read-only public access to the blobs within the container.
- **Container:** Allows public access to the container and its blobs.

7. Optionally, configure advanced settings such as metadata, default access level, retention policies, and access control.

8. Click on the "OK" or "Create" button to create the blob container.

In the following figure, we can see an overview of our Blob Container:

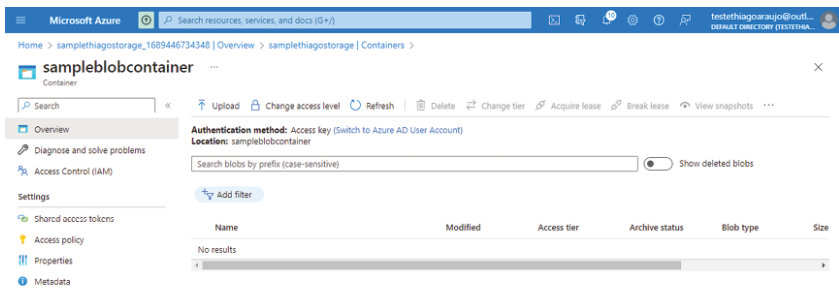


Figure 5.13: Container overview.

Connecting to the Database

In our Web Application we start by adding the following Nuget package:

Azure.Storage.Blobs

We have created the following class to be stored in the Blob:

```
1. public class SampleBlob
2. {
3.     public int Id { get; set; }
4.     public string Description { get; set; }
5.     public DateTime CreationDate { get; set; }
```

```
6. }
```

From the storage account blade, you can manage your storage resources, configure access controls, and access connection settings such as the primary connection string and access keys.

Retrieve the connection string or access keys from the Azure portal to establish a connection between your applications and the Azure Storage account.

In the following figure, we can see the Connection String from the Storage Account:

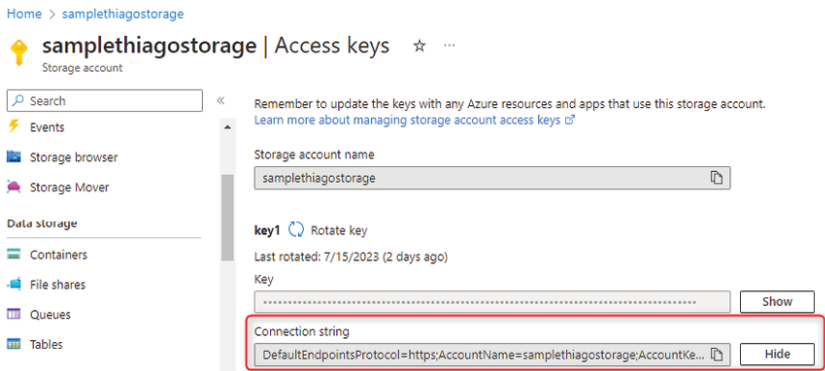


Figure 5.14: Azure Storage Account Access Keys.

We had updated the **Index.cshtml.cs** to store this new blob:

```
1. using Azure.Storage.Blobs.Models;
2. using Azure.Storage.Blobs;
3. using Microsoft.AspNetCore.Mvc;
4. using Microsoft.AspNetCore.Mvc.RazorPages;
5. using BlobStorageWebApp.Database;
6.
7. namespace BlobStorageWebApp.Pages
8. {
9.     public class IndexModel : PageModel
10.    {
11.        private readonly
ILogger<IndexModel> _logger;
```

```

12.
13.         public
IndexModel(ILogger<IndexModel> logger)
14.         {
15.             _logger = logger;
16.         }
17.
18.         public async void OnGet ()
19.         {
20.             await CreateBlobAsync ();
21.         }
22.         public async Task CreateBlobAsync ()
23.         {
24.             // Get a reference to the container
25.             BlobContainerClient container =
new BlobContainerClient (
26. "DefaultEndpointsProtocol=https;"+
27. "AccountName=samplethiagostorage;"+
28. "AccountKey=beSfcQhP9KcDeznMNlyLXgEJyDMXrgjaF9+Ar
29. +"uUbdMcUQp5yQutGWm7tKjBIm33o4G
+AStn05yfA==;"+
30. "EndpointSuffix=core.windows.net",
"sampleblobcontainer");
31.
32.             await
container.UploadBlobAsync ("sampleBlob",
33. BinaryData.FromObjectAsJson<SampleBlob> (
34. new SampleBlob
35.         { Id = 0, Description =
"testing description"
36. , CreationDate = DateTime.Now }));
37.         }
38.     }
39. }

```

This is the output after running the project:

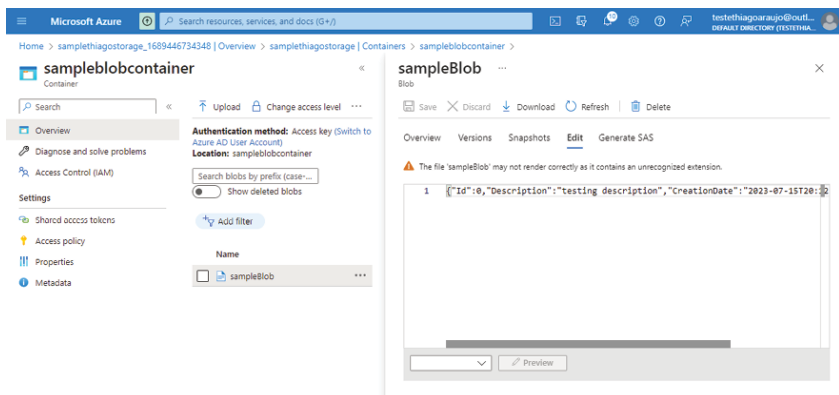


Figure 5.15: Container with recently created data.

Conclusion

In this chapter, we delved into the world of Azure SQL, Cosmos DB, and Blob Storage, three fundamental components of Azure's data services. We began by understanding the capabilities and benefits of Azure SQL, a fully managed relational database service that provides scalability, security, and ease of use. Through various usage examples, we witnessed how Azure SQL caters to a wide range of applications, from small-scale projects to enterprise-level solutions.

Next, we explored Cosmos DB, a globally distributed, multi-model database service that offers impressive scalability and low-latency performance. We learned about its versatility in supporting various data models and its global reach, making it suitable for applications requiring responsiveness and availability across multiple regions. The usage examples highlighted the potential of Cosmos DB in different domains, showcasing its ability to handle massive-scale applications seamlessly.

Lastly, we discussed Blob Storage, Azure's storage solution for unstructured data. With its scalable and cost-effective options, Blob Storage offers a convenient way to manage and access unstructured data such as images, videos, and documents. The practical usage examples demonstrated the utility of Blob Storage in diverse scenarios, emphasizing its importance in modern data management strategies.

By exploring Azure SQL, Cosmos DB, and Blob Storage in this chapter, you have gained a solid understanding of their features, benefits, and real-world applications. You are now equipped with the knowledge and skills necessary to leverage these services effectively in the Azure cloud environment. As you continue your journey with Azure, remember to consider the specific requirements of your projects and apply best practices to ensure optimal utilization of these powerful data services.

Azure SQL, Cosmos DB, and Blob Storage provide a robust foundation for building scalable, secure, and globally accessible applications. Whether you are working on a small project or managing a large enterprise solution, these services offer the flexibility and reliability needed to meet your data storage and management needs. Embrace the possibilities and continue exploring the vast capabilities of Azure's data services in your future endeavors.

In the upcoming chapter, we venture into the dynamic realm of *Async Operations with Azure Service Bus*. This topic unfolds a crucial facet of Azure's ecosystem, exploring how asynchronous operations enhance efficiency and scalability in distributed systems. We delve into the functionalities of Azure Service Bus, a powerful messaging service, and elucidate how it facilitates seamless communication between decoupled components. Through practical insights and examples, readers will gain a comprehensive understanding of how Azure Service Bus empowers applications to perform asynchronous operations, fostering resilience and responsiveness in modern cloud architectures.

CHAPTER 6

Unleashing the Power of Async Operations with Azure Service Bus

Introduction

In today's dynamic landscape, where speed, scalability, and efficient communication are paramount, mastering asynchronous operations is indispensable. This chapter immerses us in the realm of Azure Service Bus, a robust cloud-based messaging service from Microsoft Azure, offering an array of features crucial for effective asynchronous operations.

We commence with an exploration of Queues, fundamental to Azure Service Bus, where messages follow a **First-In-First-Out (FIFO)** approach. Queues ensure reliable message delivery, enhancing system resilience. We delve into creating Queues, sending/receiving messages, and navigating various message processing scenarios.

Transitioning to Topics, we discover their power in enabling a

publish/subscribe pattern, facilitating efficient broadcasting to multiple subscribers. Topics offer scalability and flexibility, empowering us to create and define Subscriptions, manage message filtering, and customize routing.

A comprehensive case study guides us through the step-by-step implementation of Azure Service Bus, offering hands-on experience and real-world insights. We progress to advanced features, exploring message sessions that streamline interactions with diverse clients. Message sessions aid in managing complex workflows and maintaining message order, optimizing system performance.

The chapter culminates in addressing error handling and exception management—a critical aspect in the realm of distributed systems. Strategies for handling exceptions, retrying message processing, and implementing robust error handling mechanisms are discussed. By the chapter's end, readers will possess a profound understanding of Azure Service Bus, equipped with the skills to build resilient systems capable of handling large workloads and facilitating seamless communication between components. This journey promises to unlock the full potential of asynchronous operations with Azure Service Bus.

Structure

This chapter covers the following topics:

- Async operations with Service Bus
- Azure Service Bus Topics
- Azure Service Bus Subscriptions
- Case study

Objectives

By the end of this chapter, you will understand the core concepts of Azure Service Bus, including Queues, Topics, and Subscriptions. You will learn how to create and manage Queues in Azure Service Bus for reliable message delivery and processing. Explore the benefits and capabilities of Topics and Subscriptions, enabling efficient publish/subscribe messaging patterns. Gain practical insights through a step-by-step case study that demonstrates the

implementation of Azure Service Bus. We will create Topics, define Subscriptions, and explore message filtering and routing techniques. Discover the use of message sessions to handle interactions with different clients and maintain message order within each session. Explore advanced features and optimizations to enhance the performance and scalability of Azure Service Bus. Understand exception management and error handling strategies to ensure the resilience and stability of your applications. Learn best practices for handling exceptions, retrying message processing, and implementing robust error handling mechanisms. The readers will acquire the knowledge and skills necessary to design and build efficient and scalable systems using Azure Service Bus for asynchronous operations.

Async operations with Service Bus

Azure Service Bus is a cloud-based messaging service provided by Microsoft Azure. It offers a robust and scalable platform for building distributed applications and systems that require asynchronous and decoupled communication between components. Azure Service Bus provides several messaging patterns, including Queues, Topics, and Subscriptions, allowing developers to design flexible and reliable messaging solutions.

Key features and components of Azure Service Bus include:

- **Queues:** Azure Service Bus Queues provide a reliable and asynchronous messaging mechanism. They follow the FIFO principle, ensuring that messages are processed in the order they are received. Queues offer reliable message delivery, message durability, and support for both one-way and request-response communication patterns.
- **Topics and Subscriptions:** Azure Service Bus Topics allow for a publish/subscribe messaging pattern. Messages published to a Topic can be received by multiple subscribers, known as Subscriptions. Subscriptions can have specific filters to receive only the relevant subset of messages, enabling efficient message distribution and allowing components to subscribe to specific message types or Topics of interest.
- **Relays:** Azure Service Bus Relays provide a way to expose on-premises services to the cloud or enable communication between on-premises and cloud-based services securely.

Relays use a bi-directional communication channel that allows clients to connect to a relay endpoint and communicate with the service behind it.

- **Message sessions:** Azure Service Bus supports message sessions, which allow related messages to be grouped together and processed sequentially. Message sessions are useful for scenarios where maintaining order and processing related messages together is critical, such as in workflows or transactions involving multiple steps.
- **Dead-lettering:** Azure Service Bus includes dead-lettering capabilities, which move messages that cannot be processed or have exceeded the maximum number of delivery attempts to a separate dead-letter Queue or Topic. Dead-lettering provides a mechanism for handling failed or poisoned messages separately, enabling developers to diagnose and resolve issues effectively.
- **Message transactions:** Azure Service Bus supports message transactions, allowing multiple messages to be sent and processed as part of a single atomic operation. This ensures that all messages are either successfully processed or rolled back in case of failure, providing consistency and reliability in message processing.
- **Auto-scaling and high availability:** Azure Service Bus is designed to be highly available and scalable. It offers automatic load balancing and partitioning of messages across multiple nodes, ensuring high throughput and fault tolerance. Azure Service Bus also provides auto-scaling capabilities to adjust resources dynamically based on workload demands.
- **Management and monitoring:** Azure Service Bus provides comprehensive management and monitoring capabilities through Azure portal, Azure CLI, PowerShell, and REST APIs. These features allow developers to monitor message throughput, manage entities like Queues and Topics, and configure various aspects of message handling and processing.

Azure Service Bus is widely used in various scenarios, including enterprise messaging, distributed systems, **Internet of Things (IoT)** messaging, event-driven architectures, and hybrid cloud integration. Its flexibility, scalability, and reliability make it an essential component for building robust and responsive applications that require asynchronous communication and seamless integration

between different components.

Azure Service Bus Queues

Azure Service Bus Queues are a core component of Azure Service Bus, providing a reliable and asynchronous messaging mechanism for building distributed applications. Queues enable decoupled communication between components, allowing them to exchange messages in a reliable and ordered manner.

Key features and characteristics of Azure Service Bus Queues include:

- **Reliable message delivery:** Azure Service Bus Queues ensure reliable message delivery by following the FIFO principle. Messages sent to a Queue are stored and processed in the order they were received, ensuring message integrity and preserving the sequence of operations.
- **Asynchronous communication:** Queues enable asynchronous communication between sender and receiver components. The sender component can push messages to a Queue without waiting for a response, and the receiver component can retrieve messages from the Queue when it is ready to process them. This asynchronous pattern allows for loose coupling and improves system responsiveness.
- **Message durability:** Messages stored in Azure Service Bus Queues are durable. They are stored redundantly across multiple nodes within the Azure infrastructure, ensuring high availability and protection against data loss. Even if a component fails or a temporary network issue occurs, messages remain in the Queue until they are successfully processed.
- **Scalability:** Azure Service Bus Queues are designed to handle high message throughput and scale seamlessly. They support automatic load balancing and partitioning of messages across multiple nodes, allowing for horizontal scalability and accommodating increased workloads.
- **One-way and request-response communication:** Queues support both one-way messaging and request-response patterns. In one-way messaging, the sender component pushes messages to the Queue without expecting a response. In request-response scenarios, the sender places a request

message in the Queue and awaits a response message from the receiver component.

- **Message Time-to-Live (TTL):** Azure Service Bus Queues provide the ability to set a **Time-to-Live (TTL)** value for messages. The TTL determines how long a message can remain in the Queue before it expires and is automatically removed. This feature helps in managing message retention and ensures that messages are processed within a specified timeframe.
- **Dead-lettering:** Azure Service Bus Queues include dead-lettering capabilities. If a message cannot be successfully processed or exceeds the maximum number of delivery attempts, it can be moved to a separate dead-letter Queue for further analysis and handling. Dead-lettering allows for effective error handling and troubleshooting.
- **Management and monitoring:** Azure Service Bus Queues can be managed and monitored through Azure portal, Azure CLI, PowerShell, and REST APIs. These tools provide capabilities to create and configure Queues, monitor message throughput, track Queue metrics, and manage access control and security settings.

Azure Service Bus Queues are widely used in various scenarios, such as task offloading, workload balancing, event-driven architectures, and distributed systems. Their reliable and asynchronous nature makes them suitable for building robust and scalable applications that require decoupled and ordered message processing.

Session Queues

Session Queues are a specialized feature within Azure Service Bus that allows related messages to be grouped together and processed sequentially by the same receiver. In regular Queues, messages are processed independently without any inherent relationship between them. However, session Queues enable message sessions, where a logical stream of messages is associated with a specific session ID.

Session Queues are particularly useful in scenarios where maintaining the order and grouping of messages is critical, such as multi-step workflows, transactional processing, or when different parts of a task need to be processed in sequence by the same receiver.

By leveraging session Queues, developers can ensure that messages

within a session are processed in the order they were sent, and the state of the session can be maintained across multiple message processing cycles. This ensures consistent and reliable handling of related messages and provides greater control over complex workflows and processing scenarios.

Azure Service Bus Topics

Azure Service Bus Topics are a powerful messaging feature provided by Azure Service Bus. Topics enable a publish/subscribe messaging pattern, allowing messages to be published once and received by multiple subscribers. Topics provide a flexible and scalable solution for broadcasting messages to interested parties.

Key features and characteristics of Azure Service Bus Topics include:

- **Publish/Subscribe messaging:** Azure Service Bus Topics follow the publish/subscribe messaging pattern. Messages published to a Topic are not sent directly to subscribers but are instead stored in the Topic. Subscribers can then create Subscriptions to the Topic and receive messages based on their specific criteria or interests. This decoupled model allows for efficient message distribution and flexible communication between components.
- **Message filtering:** Topics support message filtering based on user-defined properties. Subscriptions can define filtering rules to receive only messages that match specific criteria. This feature enables fine-grained control over which messages are delivered to subscribers, ensuring that each subscriber receives relevant and targeted messages.
- **Message routing:** Azure Service Bus Topics provide message routing capabilities. Publishers can define routing rules that determine how messages are distributed among different Subscriptions based on message properties. This allows for intelligent message routing and selective delivery to appropriate subscribers.
- **Multiple Subscriptions:** Topics support multiple Subscriptions, allowing different components or subscribers to receive messages independently. Each Subscription maintains its own cursor, enabling subscribers to receive messages at their own pace and manage their message processing independently.

- **Ordered message delivery:** Within each Subscription, messages are delivered in the order they were published. This ensures that subscribers receive messages in a sequential and consistent manner, making Topics suitable for scenarios that require ordered message processing.
- **Message sessions:** Topics also support message sessions, which enable related messages to be grouped together and processed sequentially. Message sessions are useful in scenarios where maintaining message order and processing related messages together is important, such as multi-step workflows or transactions involving multiple messages.
- **Scalability and performance:** Azure Service Bus Topics are designed to be highly scalable and performant. They support automatic load balancing and partitioning of messages across multiple nodes, allowing for high throughput and horizontal scalability. Topics can handle large message volumes and support a high number of Subscriptions.
- **Management and monitoring:** Azure Service Bus Topics can be managed and monitored through Azure portal, Azure CLI, PowerShell, and REST APIs. These tools provide capabilities to create and configure Topics and Subscriptions, monitor message throughput and Subscription activity, and manage access control and security settings.

Azure Service Bus Topics are widely used in various scenarios, including event-driven architectures, notifications, content distribution, and microservices communication. They provide a flexible and scalable solution for broadcasting messages to multiple subscribers, ensuring efficient and decoupled communication between components.

Azure Service Bus Subscriptions

Azure Service Bus Subscriptions are a key component of Azure Service Bus Topics. Subscriptions enable receiving and processing messages from a Topic based on specific criteria or interests. They provide a flexible and targeted way for subscribers to consume messages published to a Topic.

Key features and characteristics of Azure Service Bus Subscriptions include:

- **Message filtering:** Subscriptions allow for message filtering

based on user-defined properties. By specifying filtering rules, subscribers can receive only the messages that match their criteria or interests. This filtering capability enables subscribers to consume relevant messages and avoid processing unnecessary messages.

- **Selective message delivery:** Azure Service Bus Subscriptions provide selective message delivery. Each Subscription defines its own set of filtering rules, allowing subscribers to receive only the messages that match their criteria. This ensures that subscribers receive a customized subset of messages from the Topic.
- **Ordered message delivery:** Messages delivered to a Subscription within an Azure Service Bus Topic maintain their order. This means that messages are delivered and processed in the same order as they were published. Ordered message delivery ensures consistency and helps subscribers maintain the sequence of operations.
- **Multiple Subscriptions:** Topics support multiple Subscriptions, allowing different components or subscribers to receive messages independently. Each Subscription maintains its own cursor, enabling subscribers to receive messages at their own pace and manage their message processing independently.
- **Message sessions:** Subscriptions also support message sessions, which enable related messages to be grouped together and processed sequentially. Message sessions are useful in scenarios where maintaining message order and processing related messages together is important, such as multi-step workflows or transactions involving multiple messages.
- **Auto-scaling and performance:** Azure Service Bus Subscriptions are designed to be scalable and performant. They can handle high message throughput and support a high number of subscribers. Subscriptions benefit from the automatic load balancing and partitioning capabilities of Azure Service Bus, ensuring efficient message distribution.
- **Management and monitoring:** Azure Service Bus Subscriptions can be managed and monitored through the Azure portal, Azure CLI, PowerShell, and REST APIs. These tools provide capabilities to create and configure Subscriptions, define filtering rules, monitor Subscription

activity and message throughput, and manage access control and security settings.

Azure Service Bus Subscriptions are commonly used in scenarios that involve the publish/subscribe messaging pattern, event-driven architectures, notifications, and content distribution. They provide a flexible and targeted way for components to consume messages published on a Topic based on specific criteria, enabling decoupled and efficient communication between components.

Azure Service Bus vs Azure Queue Storage Queues

Azure Service Bus and Azure Queue Storage Queues serve as message brokers, facilitating communication between different components in distributed systems. However, they have distinct features and capabilities that make them suitable for different scenarios as we can see in the following table:

	Azure Service Bus	Azure Queue Storage
Communication Patterns	Publish/Subscribe, Point-to-Point	Point-to-Point
Message Ordering	Supported	Not Supported
Message Filtering	Supported	Not Supported
Dead-letter Queues	Supported	Not Supported
Scheduled Delivery	Supported	Not Supported
Message Sessions	Supported	Not Supported
Supportability	High	Medium
Supported Tl	Supported	Not Supported
Message Transactions	Supported	Not Supported
Highly reliable persistence	Yes	No
Advanced features and flexibility	Yes	No
Medium/high complexity	Yes	No

Table 6.1: Functionalities overview from Service Bus vs Queue Storage Queue

Azure Service Bus is a powerful and feature-rich messaging service that supports both Publish/Subscribe and Point-to-Point communication patterns. It provides advanced features like message filtering, Dead-letter Queues for handling failed messages, scheduled delivery of messages, message sessions for grouping related messages, and support for message transactions. Azure Service Bus is recommended for medium to high complexity scenarios where sophisticated message handling and communication patterns are required.

On the other hand, Azure Queue Storage Queues are more basic message brokers, primarily designed for Point-to-Point communication. While they provide reliable message delivery and support message ordering, they lack some of the advanced features available in Azure Service Bus, such as message filtering, scheduled delivery, and message sessions. Azure Queue Storage Queues are more suitable for low complexity scenarios where basic message queuing and processing are sufficient.

In summary, Azure Service Bus is a robust choice for scenarios with medium to high complexity and demanding messaging requirements, while Azure Queue Storage Queues offer a simpler solution for low complexity scenarios with straightforward message queuing needs. Choosing between the two depends on the specific requirements and complexity of the application or system being developed.

Case study

In the case study, we are working with various functionalities of Azure Service Bus and implementing different components to achieve specific tasks related to message handling and processing. For a better understanding we are having one console application for each consumer, also one console application for the publisher. Here is a breakdown of the tasks involved in the case study:

- Publishing messages
 - Implement a sender component to publish messages to Azure Service Bus Queues.
 - Use the Azure Service Bus SDK for .NET to send messages to the Queue.
- Consuming messages
 - Implement a receiver component to consume and process messages from the Azure Service Bus Queue.
 - Handle message processing and implement any necessary business logic.
- Consuming message batches
 - Implement a receiver component to handle messages in batches.

- Improve message processing efficiency by processing messages in batches.
- Message processor
 - Implement a message processor component responsible for handling message processing logic.
 - The message processor will be used by the receiver to process individual messages or message batches.
- Consuming sessions
 - Implement a receiver component to consume and process messages from Azure Service Bus sessions.
 - Implement logic to group related messages together using session IDs.
- Session processor
 - Implement a session processor component responsible for handling session-based message processing.
 - The session processor will be used by the receiver to process messages within a session in the order they were sent.
- Consuming Topics and Subscriptions
 - Implement subscriber components to consume and process messages from Azure Service Bus Topics and Subscriptions.
 - Define multiple Subscriptions based on different criteria or interests and implement logic for selective message delivery.
- Nuget package used:
 - Azure.Messaging.ServiceBus

Creating a new Azure Service Bus

The first step for our practical study case is to create a new Azure Service Bus component, we have to do as follows:

1. **Sign in to the Azure Portal:** Go to <https://portal.azure.com> and sign in with your Microsoft Azure account.
2. **Create a New Resource:** Once you are logged in, click on the "Create a resource" button (+) located on the left-hand

side of the Azure portal.

3. **Search for "Service Bus":** In the search bar, type "**Service Bus**" and select "**Service Bus**" from the list of available services.
4. **Click on "Create":** On the Service Bus page, click on the "**Create**" button to start the creation process.
5. **Basic details:**
 - **Subscription:** Choose the Azure Subscription where you want to create the Service Bus namespace.
 - **Resource Group:** Select an existing resource group or create a new one to contain the Service Bus namespace and related resources.
 - **Namespace Name:** Enter a unique name for your Service Bus namespace. The name must be globally unique across all Azure Service Bus namespaces.
 - **Region:** Choose the region where you want your Service Bus namespace to be deployed. Select a region that is geographically close to your applications or services for optimal performance.
 - **Pricing Tier:** Select the pricing tier that meets your requirements. Azure Service Bus offers different tiers with varying features and pricing options.
 - **Virtual Network:** If needed, you can choose to enable Virtual Network rules to restrict access to the Service Bus namespace to specific virtual networks.
 - **Tags:** Optionally, you can add tags to help organize and categorize your Azure resources.
13. **Review + Create:** Review your configuration settings and click on the "**Create**" button to start the deployment process.

The deployment of the Azure Service Bus namespace may take a few minutes. Once the deployment is complete, you will see the new Azure Service Bus namespace listed in the Azure portal. You can now start using the Service Bus for message queuing, publish/subscribe messaging, and other messaging patterns to build distributed applications. After successful deployment, our Service Bus Namespace must look like the image below:

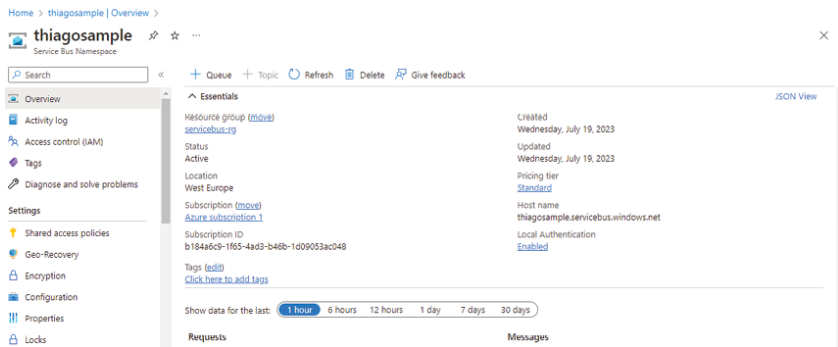


Figure 6.1: Service Bus Namespace in Azure portal

Now we have to get the connection string of our Service Bus, please follow the following steps:

1. **Access the Shared Access Policies:** Inside the Service Bus namespace, navigate to the left-hand menu, under "Settings," click on "Shared access policies."
2. **Create or Access an Existing Shared Access Policy:** By default, there will be a "RootManageSharedAccessKey" policy, which has full access permissions. While this policy can be used to obtain the connection string, it is recommended to create a custom policy with specific permissions for your application.
3. **Create a Custom Shared Access Policy (Optional):** If you want to create a custom policy, click on "Add" to create a new shared access policy. Give it a name and specify the required permissions (for example, Send, Listen, Manage) based on your application's needs.
4. **Get the Connection String:** After creating or selecting the desired shared access policy, click on the policy name to view its details. The connection string will be available in the "Primary Connection String" field. This connection string contains the necessary information for your application to authenticate and connect to the Azure Service Bus namespace, as you can see in the image below:

SAS Policy: RootManageSharedAccessKey



Save Discard Delete Regenerate Primary Key ...

☒ Manage

☐ Send

☐ Listen

Primary Key

e68b5GnhuadJfa1W49kCxfBztioIAHXU+ASbH7kjrc=

Secondary Key

ef3l7W80//Xwujl7oKkRkuWBn+zi66iq+ASbHQB3pY=

Primary Connection String

Endpoint=sb://thiagosample.servicebus.windows.net;/SharedAccessKeyName=RootMa...

Secondary Connection String

Endpoint=sb://thiagosample.servicebus.windows.net;/SharedAccessKeyName=RootMa...

SAS Policy ARM ID

/subscriptions/b184a6c9-1f65-4ad3-b46b-1d09053ac048/resourcegroups/servicebus-r...

Figure 6.2: Service Bus Connection String

Now we have to create the Queue, proceed as follows:

1. Inside the Service Bus namespace, navigate to the left-hand menu, under “Entities,” click on “Queues.”
2. **Create a New Queue:** Click on the “+ Queue” button to create a new Queue.
3. **Configure the Queue:**
 - **Name:** Enter a unique name for the Queue. The name must be globally unique within the Azure Service Bus namespace.
 - **Enable Partitioning:** By default, partitioning is disabled, but you can choose to enable it for improved scalability if you expect high message throughput.
 - **Duplicate Detection:** You can enable duplicate detection to prevent the processing of duplicate messages within a specified time window.

- **Time-to-Live (TTL):** Optionally, you can set a TTL value for messages in the Queue. Messages that exceed the TTL will be automatically removed from the Queue.
8. **Access Control (Optional):** Configure access control to set permissions for different users or applications accessing the Queue.
 9. **Create the Queue:** Click on the “**Create**” button to create the Queue. The Queue will be provisioned within the Azure Service Bus namespace, as we can see in the following image:

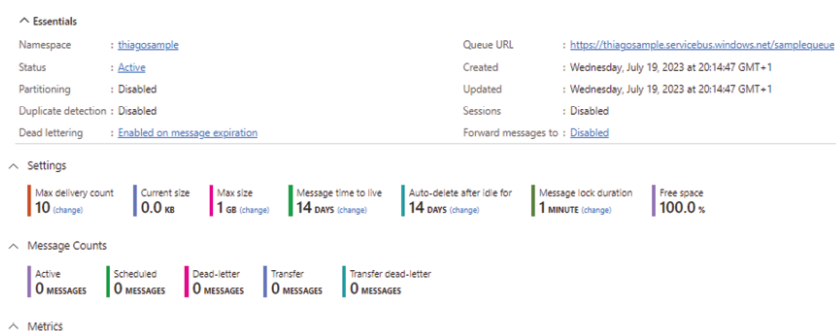


Figure 6.3: Service Bus recently created Queue

Now it is time for us to create the session queue, proceed as the following steps:

1. Inside the Service Bus Namespace, navigate to the left-hand menu, under “**Entities**,” click on “**Queues**.”
2. **Create a New Queue:** Click on the “+ **Queue**” button to create a new Queue.
3. **Configure the Queue:**
 - **Name:** Enter a unique name for the Queue. The name must be globally unique within the Azure Service Bus namespace.
 - **Enable Partitioning:** By default, partitioning is disabled, but you can choose to enable it for improved scalability if you expect high message throughput.
 - **Session:** To create a session-enabled Queue, select the “**Enable session**” checkbox. This will enable the Queue

to support message sessions.

- 7. Access Control (Optional):** Configure access control to set permissions for different users or applications accessing the Queue.
- 8. Create the Queue:** Click on the “**Create**” button to create the session-enabled Queue. The Queue will be provisioned within the Azure Service Bus namespace, as we can see in the following image:



Figure 6.4: Service Bus recently created session Queue

To create the topic, follow the next steps:

1. Inside the Service Bus namespace, navigate to the left-hand menu, under "Entities," click on "Topics."
2. **Create a new Topic:** Click on the "+ Topic" button to create a new Topic.
3. **Configure the Topic:**
 - **Name:** Enter a unique name for the Topic. The name must be globally unique within the Azure Service Bus namespace.
 - **Enable partitioning:** By default, partitioning is disabled, but you can choose to enable it for improved scalability if you expect high message throughput.
6. **Access control (optional):** Configure access control to set permissions for different users or applications accessing the Topic.
7. **Create the Topic:** Click on the "Create" button to create the Topic. The Topic will be provisioned within the Azure Service Bus namespace, as we can see in the following image:

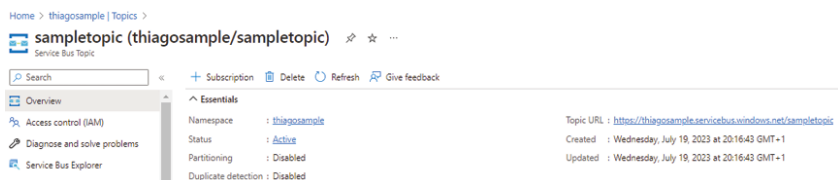


Figure 6.5: Service Bus recently created Topic

It is time for us to create the subscription, proceed with the following steps:

1. Inside the Service Bus namespace, navigate to the left-hand menu, under "**Entities**," click on "**Topics**."
2. **Select the Topic:** Click on the Topic for which you want to create the Subscription.
3. **Create a New Subscription:** Inside the selected Topic, navigate to the left-hand menu, under "**Entities**," click on "**Subscriptions**."
4. **Create a new Subscription:** Click on the "+ **Subscription**" button to create a new Subscription.
5. **Configure the Subscription:**
 - **Name:** Enter a unique name for the Subscription. The name must be unique within the selected Topic.
 - **Filtering (Optional):** Optionally, you can set up filters for the Subscription to receive only specific messages based on message properties.
8. **Access control (Optional):** Configure access control to set permissions for different users or applications accessing the Subscription.
9. **Create the Subscription:** Click on the "Create" button to create the Subscription. The Subscription will be provisioned within the selected Azure Service Bus Topic, as we can see in the following image:

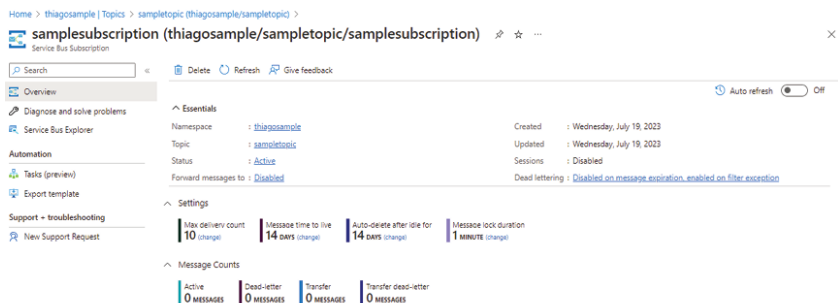


Figure 6.6: Service Bus recently created Subscription

Publishing to Azure Service Bus

The Publisher component plays a central role in our distributed application, leveraging Azure Service Bus with .NET and C# to manage crucial messaging patterns. This versatile component excels at publishing messages, transmitting message batches, working with Topics, and handling message sessions.

In this practical case study, we have the following code to use all the components created above. This code connects to the components and publish messages.

```

1. // See https://aka.ms/new-console-template for more
   information
2. using Azure.Core;
3. using Azure.Messaging.ServiceBus;
4. using
   Azure.Messaging.ServiceBus.Administration;
5. using System;
6. using System.Threading.Tasks;
7.
8. string connectionString = "Endpoint=sb://
   thiagosample.servicebus.windows.net/;SharedAccess
   +ASbH7kjrc=";
9. string queueName = "samplequeue";
10. string sessionQueueName =
    "samplesessionqueue";
11. string topicName = "sampletopic";
12.

```

```

13. // name of your Service Bus queue
14. // the client that owns the connection and can be used to
   create senders and receivers
15. // the sender used to publish messages to the queue
16.
17. await using var client = new
   ServiceBusClient(connectionString);
18. await using var normalQueueSender =
   client.CreateSender(queueName);
19. await using var sessionQueueSender =
   client.CreateSender(sessionQueueName);
20. await using var topicQueueSender =
   client.CreateSender(topicName);
21.
22. await
   normalQueueSender.SendMessageAsync(new
   ServiceBusMessage("1 message at: " +
   DateTime.Now));
23. await
   normalQueueSender.SendMessageAsync(new
   ServiceBusMessage("2 message at: " +
   DateTime.Now));
24. await
   normalQueueSender.SendMessageAsync(new
   ServiceBusMessage("3 message at: " +
   DateTime.Now));
25.
26. // create a batch
27. using ServiceBusMessageBatch messageBatch =
   await
   normalQueueSender.CreateMessageBatchAsync();
28.
29. for (int i = 1; i <= new Random().Next(10);
   i++)
30. {
31.     // try adding a message to the batch
32.     if (!messageBatch.TryAddMessage(new
   ServiceBusMessage($"Message {i} at: « +

```

```

        DateTime.Now)))
33.     {
34.         // if it is too large for the batch
35.         throw new Exception($"The message
           {i} is too large to fit in the batch.»);
36.     }
37. }
38. await
    normalQueueSender.SendMessagesAsync(messageBatch);
39.
40. await topicQueueSender.SendMessageAsync(new
    ServiceBusMessage("1 topic message at: " +
    DateTime.Now));
41. await topicQueueSender.SendMessageAsync(new
    ServiceBusMessage("2 topic message at: " +
    DateTime.Now));
42. await topicQueueSender.SendMessageAsync(new
    ServiceBusMessage("3 topic message at: " +
    DateTime.Now));
43.
44. await
    sessionQueueSender.SendMessageAsync(new
    ServiceBusMessage("1 session message at: "
    + DateTime.Now) { SessionId =
    "sampleSessionId" });
45. await
    sessionQueueSender.SendMessageAsync(new
    ServiceBusMessage("2 session message at: "
    + DateTime.Now) { SessionId =
    "sampleSessionId" });
46. await
    sessionQueueSender.SendMessageAsync(new
    ServiceBusMessage("3 session message at: "
    + DateTime.Now) { SessionId =
    "sampleSessionId" });
47. await
    sessionQueueSender.SendMessageAsync(new
    ServiceBusMessage("1 session message at: "

```

```
+ DateTime.Now) { SessionId =  
"sampleSessionId2" });
```

```
48. await  
sessionQueueSender.SendMessageAsync(new  
ServiceBusMessage("2 session message at: "  
+ DateTime.Now) { SessionId =  
"sampleSessionId2" });
```

```
49. await  
sessionQueueSender.SendMessageAsync(new  
ServiceBusMessage("3 session message at: "  
+ DateTime.Now) { SessionId =  
"sampleSessionId2" });
```

50.

```
51. Console.WriteLine("Messages Published!");
```

This is the Azure portal after publishing to the Queue and session Queues:

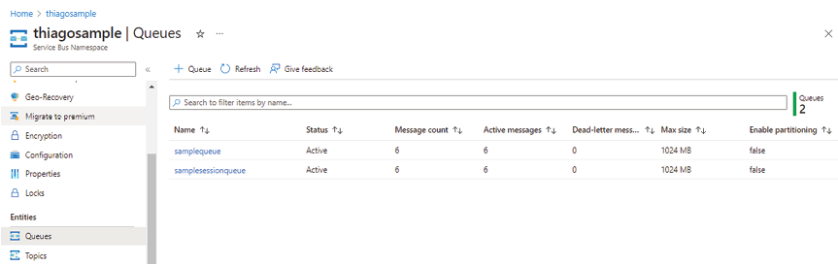


Figure 6.7: Service Bus Queues with messages

This is the Azure portal after publishing to Topic and Subscription:



Figure 6.8: Service Bus Topic with messages

Consuming messages

In this process, receiver components retrieve messages from the Queue:

```
1. // See https://aka.ms/new-console-template for more  
information  
2. using Azure.Messaging.ServiceBus;  
3.   
4. string connectionString = "Endpoint=sb://  
    thiagosample.servicebus.windows.net/;SharedAccess  
    +ASbH7kjr=";  
5. string queueName = "samplequeue";  
6.   
7. await using ServiceBusClient client = new  
    ServiceBusClient(connectionString);  
8.   
9. // create the options to use for configuring the processor  
10. var options = new  
    ServiceBusProcessorOptions  
11. {  
12.     // By default or when AutoCompleteMessages is set to  
true, the processor will complete the message after executing  
the message handler  
13.     // Set AutoCompleteMessages to false to [settle  
messages](https://docs.microsoft.com/en-us/azure/service-  
bus-messaging/message-transfers-locks-settlement#peeklock)  
on your own.  
14.     // In both cases, if the message handler throws an  
exception without settling the message, the processor will  
abandon the message.  
15.     AutoCompleteMessages = false,  
16.   
17.     // I can also allow for multi-threading  
18.     MaxConcurrentCalls = 2  
19. };  
20.   

```

```
21. // create a processor that we can use to process the messages
22. await using ServiceBusProcessor processor =
    client.CreateProcessor(queueName, options);
23.
24. // configure the message and error handler to use
25. processor.ProcessMessageAsync +=
    MessageHandler;
26. processor.ProcessErrorAsync +=
    ErrorHandler;
27.
28. async Task
    MessageHandler(ProcessMessageEventArgs
    args)
29. {
30.     string body =
    args.Message.Body.ToString();
31.     Console.WriteLine(body);
32.
33.     // we can evaluate application logic and use that to
    determine how to settle the message.
34.     await
    args.CompleteMessageAsync(args.Message);
35. }
36.
37. Task ErrorHandler(ProcessErrorEventArgs
    args)
38. {
39.     // the error source tells me at what point in the
    processing an error occurred
40.     Console.WriteLine(args.ErrorSource);
41.     // the fully qualified namespace is available
42.     Console.WriteLine(args.FullyQualifiedNamespace);
43.     // as well as the entity path
44.     Console.WriteLine(args.EntityPath);
45.     Console.WriteLine(args.Exception.ToString());
```

```

46.         return Task.CompletedTask;
47.     }
48.
49.     // start processing
50.     await processor.StartProcessingAsync();
51.
52.     // since the processing happens in the background, we add a
    Console.ReadKey to allow the processing to continue until a
    key is pressed.
53.     Console.ReadKey();

```

And in the image below we can see the output from our program:

```

Microsoft Visual Studio Debu
Hello, World!
1 message at: 21/07/2023 17:38:48
C:\Pessoal\thiago-vivas\Books\C# 12\Chapter 6\Chapter6\AzureServiceBusConsumerFour\bin\Debug\net7.0\AzSBMessageConsumer.exe (process 45680) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .

```

Figure 6.9: Console application output from consuming a message

Consuming message batches

Instead of consuming messages individually, this approach allows receiver components to handle several messages together, optimizing message processing and reducing communication overhead.

This is the code to consume message batches used in this case study:

```

1. // See https://aka.ms/new-console-template for more
   information
2. using Azure.Messaging.ServiceBus;
3.
4. string connectionString = "Endpoint=sb://
   thiagosample.servicebus.windows.net/;SharedAccess
   +ASbH7kjrc=";
5. string queueName = "samplequeue";
6.
7.
8. await using ServiceBusClient client = new

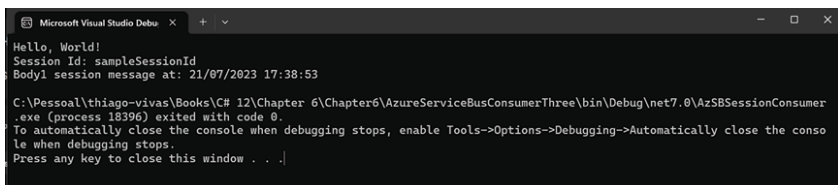
```

```

        ServiceBusClient(connectionString);
9.
10. // create a receiver that we can use to receive the messages
11. await using ServiceBusReceiver receiver =
    client.CreateReceiver(queueName);
12.
13. // the received message is a different type as it contains some
    service set properties
14. // a batch of messages (maximum of 4 in this case) are
    received
15. IReadOnlyList<ServiceBusReceivedMessage>
    receivedMessages = await
    receiver.ReceiveMessagesAsync(maxMessages:
    4);
16. Console.WriteLine("Number of messages in
    the batch: " + receivedMessages.Count);
17.
18. // go through each of the messages received
19. foreach (ServiceBusReceivedMessage
    receivedMessage in receivedMessages)
20. {
21.     // get the message body as a string
22.     string body =
    receivedMessage.Body.ToString();
23.     Console.WriteLine(body);
24. }

```

And after the successful execution of our program, we have the following output as we can see in the image:



```

Microsoft Visual Studio Debu
Hello, World!
Session Id: sampleSessionId
Body1 session message at: 21/07/2023 17:38:53

C:\Personal\things-vivas\Books\C# 12\Chapter 6\Chapter6\AzureServiceBusConsumerThree\bin\Debug\net7.0\AzSBSessionConsumer
.exe (process 18396) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .

```

Figure 6.10: Console application output from consuming message batch

Message processor

The concept of a message processor in Azure Service Bus revolves around building robust and reliable mechanisms to handle incoming messages from Queues or subscriptions. The Message processor acts as a fundamental component responsible for receiving messages, processing them, and executing the required application logic.

```
1. // See https://aka.ms/new-console-template for more information
2. using Azure.Messaging.ServiceBus;
3.
4. string connectionString = "Endpoint=sb://thiagosample.servicebus.windows.net/;SharedAccessKey=ASbH7kjrc=";
5. string queueName = "samplequeue";
6.
7. await using ServiceBusClient client = new ServiceBusClient(connectionString);
8.
9. // create the options to use for configuring the processor
10. var options = new ServiceBusProcessorOptions
11. {
12.     // By default or when AutoCompleteMessages is set to true, the processor will complete the message after executing the message handler
13.     // Set AutoCompleteMessages to false to [settle messages](https://docs.microsoft.com/en-us/azure/service-bus-messaging/message-transfers-locks-settlement#peeklock) on your own.
14.     // In both cases, if the message handler throws an exception without settling the message, the processor will abandon the message.
15.     AutoCompleteMessages = false,
16.
17.     // I can also allow for multi-threading
18.     MaxConcurrentCalls = 2
19. };
```

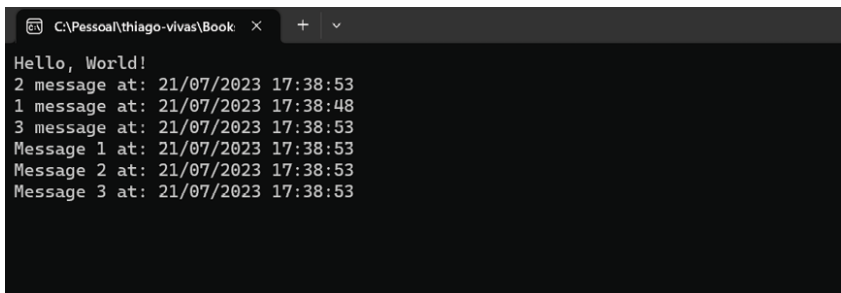
```
20.
21. // create a processor that we can use to process the messages
22. await using ServiceBusProcessor processor =
    client.CreateProcessor(queueName, options);
23.
24. // configure the message and error handler to use
25. processor.ProcessMessageAsync +=
    MessageHandler;
26. processor.ProcessErrorAsync +=
    ErrorHandler;
27.
28. async Task
    MessageHandler(ProcessMessageEventArgs
    args)
29. {
30.     string body =
    args.Message.Body.ToString();
31.     Console.WriteLine(body);
32.
33.     // we can evaluate application logic and use that to
determine how to settle the message.
34.     await
    args.CompleteMessageAsync(args.Message);
35. }
36.
37. Task ErrorHandler(ProcessErrorEventArgs
    args)
38. {
39.     // the error source tells me at what point in the
processing an error occurred
40.     Console.WriteLine(args.ErrorSource);
41.     // the fully qualified namespace is available
42.     Console.WriteLine(args.FullyQualifiedNamespace);
43.     // as well as the entity path
44.     Console.WriteLine(args.EntityPath);
45.
```

```

        Console.WriteLine(args.Exception.ToString());
46.         return Task.CompletedTask;
47.     }
48.
49. // start processing
50. await processor.StartProcessingAsync();
51.
52. // since the processing happens in the background, we add a
    Console.ReadKey to allow the processing to continue until a
    key is pressed.
53. Console.ReadKey();
54.

```

After the execution of our program, we can see our messages being processed as the following image shows:



```

C:\Pessoal\thiago-vivas\Book
Hello, World!
2 message at: 21/07/2023 17:38:53
1 message at: 21/07/2023 17:38:48
3 message at: 21/07/2023 17:38:53
Message 1 at: 21/07/2023 17:38:53
Message 2 at: 21/07/2023 17:38:53
Message 3 at: 21/07/2023 17:38:53

```

Figure 6.11: Console Application output from Message Processor

Consuming sessions

Consuming sessions in Azure Service Bus focuses on handling related messages that are grouped together into logical units known as **message sessions**. Each session contains a sequence of messages with the same session ID, allowing them to be processed in a coordinated and ordered manner.

To consume session we need the following code, this will be responsible to connect with service bus and receive the messages in the session:

1. *// See <https://aka.ms/new-console-template> for more information*
- 2.

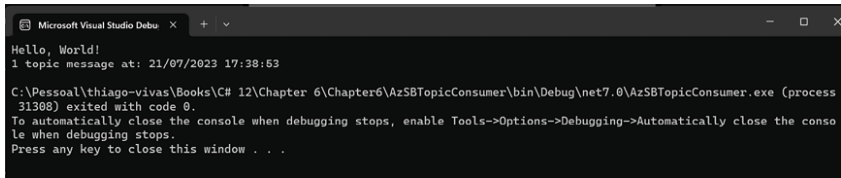
```

3. using Azure.Messaging.ServiceBus;
4.
5. string connectionString = "Endpoint=sb://
   thiagosample.servicebus.windows.net/;SharedAccess
   +ASbH7kjrc=";
6. string sessionQueueName =
   "samplesessionqueue";
7.
8. await using ServiceBusClient client = new
   ServiceBusClient(connectionString);
9.
10. // create a receiver specifying a particular session
11. await using ServiceBusSessionReceiver
   receiver = await
   client.AcceptSessionAsync(sessionQueueName,
   "sampleSessionId");
12.
13. // the received message is a different type as it contains some
   service set properties
14. ServiceBusReceivedMessage receivedMessage =
   await receiver.ReceiveMessageAsync();
15. Console.WriteLine("Session Id: " +
   receivedMessage.SessionId);
16. Console.WriteLine("Body" +
   receivedMessage.Body);
17.
18. // we can also set arbitrary session state using this receiver
19. // the state is specific to the session, and not any particular
   message
20. await receiver.SetSessionStateAsync(new
   BinaryData("brand new state"));
21.
22. // complete the message, thereby deleting it from the service
23. await
   receiver.CompleteMessageAsync(receivedMessage);

```

After successful execution of our session receiver, we have the

following output:

A screenshot of a Microsoft Visual Studio Debug Console window. The window title is "Microsoft Visual Studio Debu". The output text is: "Hello, World!", "1 topic message at: 21/07/2023 17:38:53", "C:\Pessoal\thiago-vivas\Books\CH 12\Chapter 6\Chapter6\AzSBTopicConsumer\bin\Debug\net7.0\AzSBTopicConsumer.exe (process 31308) exited with code 0.", "To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.", and "Press any key to close this window . . .".

```
Microsoft Visual Studio Debu
Hello, World!
1 topic message at: 21/07/2023 17:38:53
C:\Pessoal\thiago-vivas\Books\CH 12\Chapter 6\Chapter6\AzSBTopicConsumer\bin\Debug\net7.0\AzSBTopicConsumer.exe (process 31308) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Figure 6.12: Console application output from consuming a session

Session processor

The session processor acts as a key component responsible for processing messages within a session, ensuring sequential and coordinated operations.

The following code is responsible to create and execute the session processor:

```
1. // See https://aka.ms/new-console-template for more
   information
2. using Azure.Messaging.ServiceBus;
3.
4. string connectionString = "Endpoint=sb://
   thiagosample.servicebus.windows.net/;SharedAccess
   +ASbH7kjrc=";
5. string queueName = "samplequeue";
6. string sessionQueueName =
   "samplesessionqueue";
7. string topicName = "sampletopic";
8. string subscriptionName =
   "samplesubscription";
9.
10. // since ServiceBusClient implements IAsyncDisposable we
    create it with "await using"
11. await using ServiceBusClient client = new
    ServiceBusClient(connectionString);
12.
13. // create the options to use for configuring the processor
14. var options = new
    ServiceBusSessionProcessorOptions
```

```

15. {
16.     // By default after the message handler returns, the
    processor will complete the message
17.     // If I want more fine-grained control over settlement, I
    can set this to false.
18.     AutoCompleteMessages = false,
19.
20.     // I can also allow for processing multiple sessions
21.     MaxConcurrentSessions = 5,
22.
23.     // By default or when AutoCompleteMessages is set to
    true, the processor will complete the message after executing
    the message handler
24.     // Set AutoCompleteMessages to false to [settle
    messages](https://docs.microsoft.com/en-us/azure/service-
    bus-messaging/message-transfers-locks-settlement#peeklock)
    on your own.
25.     // In both cases, if the message handler throws an
    exception without settling the message, the processor will
    abandon the message.
26.     MaxConcurrentCallsPerSession = 2,
27.
28.     // Processing can be optionally limited to a subset of
    session Ids.
29.     SessionIds = { "sampleSessionId",
    "sampleSessionId2" },
30. };
31.
32. // create a session processor that we can use to process the
    messages
33. await using ServiceBusSessionProcessor
    processor =
    client.CreateSessionProcessor(sessionQueueName,
    options);
34.
35. // configure the message and error handler to use
36. processor.ProcessMessageAsync +=

```

```

    MessageHandler;
37. processor.ProcessErrorAsync +=
    ErrorHandler;
38.
39. async Task
    MessageHandler(ProcessSessionMessageEventArgs
    args)
40. {
41.
42.     var body =
    args.Message.Body.ToString();
43.
44.     Console.WriteLine("Session Id: " +
    args.Message.SessionId);
45.     Console.WriteLine("Body" + body);
46.
47.     // we can evaluate application logic and use that to
    determine how to settle the message.
48.     await
    args.CompleteMessageAsync(args.Message);
49.
50.     // we can also set arbitrary session state using this
    receiver
51.     // the state is specific to the session, and not any
    particular message
52.     await args.SetSessionStateAsync(new
    BinaryData("sample state"));
53. }
54.
55. Task ErrorHandler(ProcessErrorEventArgs
    args)
56. {
57.     // the error source tells me at what point in the
    processing an error occurred
58.     Console.WriteLine(args.ErrorSource);
59.     // the fully qualified namespace is available
60.

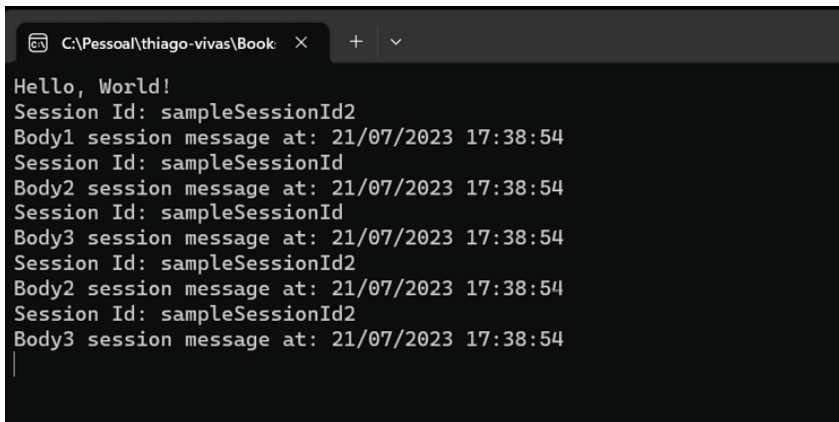
```

```

        Console.WriteLine(args.FullyQualifiedNamespace);
61.    // as well as the entity path
62.    Console.WriteLine(args.EntityPath);
63.
        Console.WriteLine(args.Exception.ToString());
64.    return Task.CompletedTask;
65. }
66.
67. // start processing
68. await processor.StartProcessingAsync();
69.
70. // since the processing happens in the background, we add a
    Console.ReadKey to allow the processing to continue until a
    key is pressed.
71. Console.ReadKey();
72.

```

After successful executing our session processor, we have the following output:



```

C:\Pessoal\thiago-vivas\Book
Hello, World!
Session Id: sampleSessionId2
Body1 session message at: 21/07/2023 17:38:54
Session Id: sampleSessionId
Body2 session message at: 21/07/2023 17:38:54
Session Id: sampleSessionId
Body3 session message at: 21/07/2023 17:38:54
Session Id: sampleSessionId2
Body2 session message at: 21/07/2023 17:38:54
Session Id: sampleSessionId2
Body3 session message at: 21/07/2023 17:38:54

```

Figure 6.13: Console application output from session processor

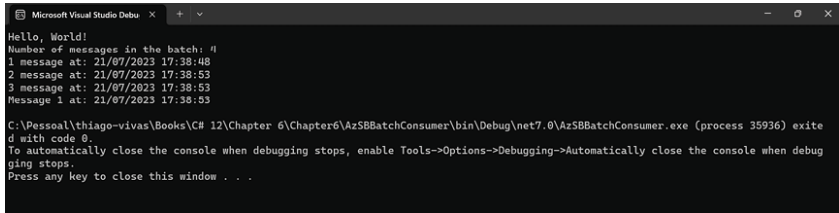
Consuming Topics and Subscriptions

This pattern enables multiple subscriber components to receive messages from a single Topic based on their specific interests or Subscription criteria.

To consume topics and subscriptions we need the following code:

```
1. // See https://aka.ms/new-console-template for more information
2. using Azure.Messaging.ServiceBus;
3.
4. string connectionString = "Endpoint=sb://thiagosample.servicebus.windows.net/;SharedAccessKey=ASbH7kjrc=";
5. string topicName = "sampletopic";
6. string subscriptionName = "samplesubscription";
7.
8.
9. await using ServiceBusClient client = new ServiceBusClient(connectionString);
10.
11. // create a receiver that we can use to receive the message// create a receiver for our subscription that we can use to receive the message
12. await using ServiceBusReceiver receiver = client.CreateReceiver(topicName, subscriptionName);
13.
14. // the received message is a different type as it contains some service set properties
15. ServiceBusReceivedMessage receivedMessage = await receiver.ReceiveMessageAsync();
16.
17. // get the message body as a string
18. string body = receivedMessage.Body.ToString();
19. Console.WriteLine(body);
20.
```

And after successfully execution of our program, we have the following output:

The image shows a screenshot of the Microsoft Visual Studio Debug Console window. The window has a title bar with the text "Microsoft Visual Studio Debug" and standard window controls (minimize, maximize, close). The console output is as follows:

```
Hello, World!  
Number of messages in the batch: 4  
1 message at: 21/07/2023 17:38:48  
2 message at: 21/07/2023 17:38:53  
3 message at: 21/07/2023 17:38:53  
Message 1 at: 21/07/2023 17:38:53  
  
C:\Personal\thiago-vivas\Books\C# 12\Chapter 6\Chapter6\AzSBBatchConsumer\bin\Debug\net7.0\AzSBBatchConsumer.exe (process 35936) exited with code 0.  
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.  
Press any key to close this window . . .
```

Figure 6.14: Console application output from consuming Topic and Subscription

In the end, this is our project solution. We have one console application per consumer, plus one console application for the publisher as we can see in the following image:

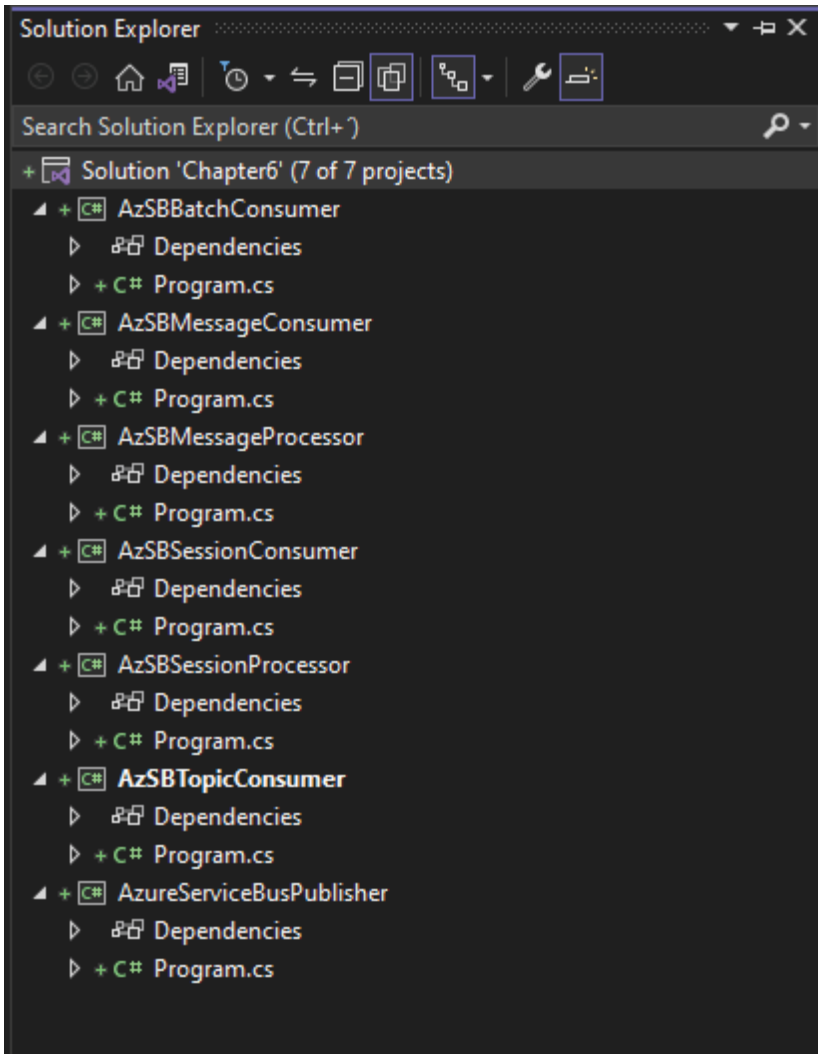


Figure 6.15: ProjectsSolution with console applications

Conclusion

In this chapter, we have explored the power and potential of Azure Service Bus for unleashing the capabilities of asynchronous operations. We began by understanding the core concepts of Queues, Topics, and Subscriptions, which form the foundation of the Azure Service Bus messaging model. We learned how to create Queues, send and receive messages, and ensure reliable message

delivery and processing.

Moving forward, we delved into Topics and Subscriptions, enabling us to implement a publish/subscribe pattern. We discovered how to create Topics, define Subscriptions, and leverage message filtering and routing techniques to efficiently distribute messages to interested parties. This flexibility and scalability are essential for building responsive and decoupled systems.

The case study provided valuable hands-on experience, guiding us through the step-by-step implementation of Azure Service Bus. We created a new Azure Service Bus instance, defined Topics, Subscriptions, and notifications, and gained practical insights into real-world scenarios. This case study equipped us with the knowledge and skills required to apply Azure Service Bus effectively in our own projects.

We also explored the use of message sessions, which allowed us to handle interactions with different clients and maintain message order within each session. Message sessions proved to be a valuable tool for managing complex workflows and optimizing performance in distributed systems.

Finally, we discussed exception management and error handling strategies, acknowledging the importance of being prepared for unexpected scenarios and failures. We learned best practices for handling exceptions, retrying message processing, and implementing robust error handling mechanisms to ensure the resilience and stability of our applications.

By mastering Azure Service Bus and its features, you are now equipped to design and build efficient and scalable systems that leverage asynchronous operations. Whether it is handling large workloads, decoupling components, or ensuring reliable message delivery, Azure Service Bus offers the tools and capabilities you need.

As you continue your journey, remember to keep exploring the vast array of features and optimizations available within Azure Service Bus. Stay up-to-date with the latest updates and best practices to maximize the potential of this powerful messaging service.

Embrace the power of Azure Service Bus and unlock the full potential of asynchronous operations in your applications. With the

knowledge gained in this chapter, you are well-prepared to design, implement, and optimize systems that can handle complex workflows, distribute tasks seamlessly, and facilitate efficient communication.

In the upcoming chapter, we delve into the critical domain of "Azure Key Vault." This topic unfolds as a cornerstone in Azure's security and identity management framework. We explore the pivotal role of Azure Key Vault in safeguarding sensitive information such as cryptographic keys, secrets, and certificates. Readers will gain insights into the principles of secure key management, encryption, and the seamless integration of Azure Key Vault into cloud applications. Through practical examples and use cases, this chapter aims to equip readers with the knowledge to enhance the security posture of their Azure-based solutions by leveraging the robust capabilities of Azure Key Vault.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 7

Securing Your Apps with Azure Key Vault

Introduction

In today's interconnected world, the security of our applications and sensitive data is of utmost importance. With the increasing number of cyber threats and data breaches, it has become essential to adopt robust security measures to protect our valuable assets. Azure Key Vault is a powerful cloud-based service provided by Microsoft Azure that allows you to safeguard cryptographic keys, secrets, and certificates used by your applications.

In this chapter, we will discuss securing your applications with Azure Key Vault. We will begin with an overview of Azure Key Vault, exploring its capabilities and benefits. You will gain a comprehensive understanding of why Azure Key Vault is a crucial component in your application security strategy.

Authentication is a vital aspect of any secure system, and Azure Key Vault provides various authentication mechanisms to ensure only authorized access. We will explore these authentication options in detail, discussing how to configure and manage them effectively.

Access policies play a significant role in controlling who can perform what actions on the resources within Azure Key Vault. We will delve into the intricacies of access policies, learning how to define granular permissions and manage them to maintain a strong security posture.

To provide a practical perspective, we will present a case study that demonstrates the step-by-step implementation of Azure Key Vault in a real-world scenario. You will witness how to create an Azure Key Vault, define and manage access policies, and leverage its features to secure your applications effectively.

Finally, we will showcase an example of accessing a key stored in Azure Key Vault through a .NET console application. You will learn how to integrate your applications with Azure Key Vault, ensuring secure retrieval and utilization of keys.

By the end of this chapter, you will have gained the knowledge and skills necessary to secure your applications using Azure Key Vault. You will be equipped with the tools to protect your cryptographic keys, secrets, and certificates, enabling you to build robust and secure applications that safeguard your sensitive data. So, let us embark on this journey to fortify your applications and defend against potential security threats with Azure Key Vault.

Structure

This chapter covers the following topics:

- Azure Key Vault Overview
- Azure Key Vault Authentication
- Azure Key Vault access policies
- Case study
 - Creating Azure Key Vault
 - Managing Key Vault Access policies
 - Accessing a key through a .Net Web Application

Objectives

This chapter equips you with a solid understanding of Azure Key Vault's core concepts, highlighting its pivotal role in securing

applications. Navigate diverse authentication mechanisms within Azure Key Vault, optimizing configurations for efficacy. Master access policies to define granular permissions, and follow a case study for practical implementation in real-world scenarios. Seamlessly integrate a .NET console application with Azure Key Vault to securely access keys. This chapter empowers you with practical skills and best practices to fortify your applications, enhancing confidence in crafting resilient security strategies. Apply these insights to bolster the security of your applications with Azure Key Vault.

Azure Key Vault Overview

Azure Key Vault is a cloud-based vault service offered by Microsoft Azure, enabling you to secure sensitive data from your project, like connection strings, certificates, and cryptographic keys. With Key Vault you may share the same sensitive information among different applications without this information being visible by any developer.

Key Vault main features and benefits are listed below:

- **Centralized key management:** Azure Key Vault provides stores your keys, secrets, and certificates in a centralized location, facilitating different applications to access the same sensitive information.
- **Secure storage:** Key Vault ensures that your keys, secrets, and certificates are secure, not being able to be accessed by anyone without permission.
- **Key lifecycle management:** Key Vault allow you to generate, import, rotate and delete keys, making it easier to maintain your keys.
- **Secrets management:** Application connection strings, API keys, and passwords are considered secrets, and you can store them in Azure Key Vault to facilitate its management and security.
- **Access control:** Key Vault offers access control via access policies. You can set the access policies for your different users, applications, or groups by having specific policies managing the access to your keys, secrets, and certificates. The access control follows the principle of the least privilege,

helping to restrict access to your sensitive data.

- **Auditing and monitoring:** Key Vault logs all interactions and operations performed on keys, secrets, and certificates. This allows you to monitor and track access, ensuring compliance and providing an audit trail for security incidents.
- **Integration with Azure Services:** Azure Key Vault seamlessly integrates with other Azure services, such as Azure Functions, Azure App Service, and Azure Virtual Machines. This allows you to securely retrieve and use keys and secrets within your applications without exposing them in your code.
- **Developer-friendly APIs:** Key Vault offers REST APIs, Azure SDKs, and client libraries for various programming languages, making it easy for developers to interact with and utilize its features in their applications.

By leveraging Azure Key Vault, you can enhance the security of your applications by protecting sensitive information, simplifying key management, and enforcing strict access control policies. It provides a robust and scalable solution for safeguarding cryptographic keys, secrets, and certificates, enabling you to build secure and trusted applications in the Azure ecosystem.

Azure Key Vault Authentication

Azure Key Vault offers multiple authentication mechanisms to ensure secure access to the stored keys, secrets, and certificates. These authentication methods provide different options for authenticating and authorizing users or applications attempting to access Azure Key Vault resources. Here are some of the key authentication mechanisms available:

- **Azure Active Directory (Azure AD) Authentication:**
 - Azure AD authentication is the recommended method for authenticating users and applications to Azure Key Vault.
 - It leverages Azure AD identities and access tokens to grant access to Key Vault resources.
 - Users or applications authenticate using their Azure AD credentials, and Azure Key Vault verifies their identity using Azure AD tokens.
 - This authentication method allows for fine-grained access

control and supports **role-based access control (RBAC)**.

- **Shared Access Signatures (SAS) Authentication:**

- SAS authentication allows clients to authenticate directly with Azure Key Vault without relying on Azure AD.
- It involves generating a SAS token with specific permissions and a validity period.
- The SAS token is appended to the Key Vault URL, allowing the client to authenticate and perform authorized operations.
- This authentication method is useful for scenarios where direct authentication with Azure AD is not possible or desired.

- **Managed identity authentication:**

- Azure Key Vault supports Azure Managed Identity, which eliminates the need for managing and storing credentials.
- Managed Identity enables applications deployed on Azure services (for example, Azure Virtual Machines, Azure App Service) to obtain an identity.
- The application's identity is then used to authenticate with Azure Key Vault, and access can be granted based on the application's identity.

- **Service principal authentication:**

- Service principal authentication allows applications to authenticate using a service principal, which is a non-human identity.
- A service principal represents the application and is assigned specific permissions to access Key Vault resources.
- This authentication method is commonly used when applications or services need to authenticate programmatically without user interaction.

- **Certificate authentication:**

- Certificate authentication involves using an X.509 certificate to authenticate an application or user.
- The certificate is associated with the Azure AD application or user, and the private key is used for authentication.
- This method is suitable for scenarios where strong client

authentication is required.

By offering these authentication mechanisms, Azure Key Vault provides flexibility and options for securing access to the stored keys, secrets, and certificates. You can choose the appropriate authentication method based on your application's requirements, security needs, and the type of client interacting with Key Vault resources.

Azure Key Vault Access policies

Azure Key Vault access policies are an integral part of controlling and managing access to Key Vault resources. Access policies define the permissions granted to users, applications, or security principals, allowing them to perform specific operations on keys, secrets, and certificates stored in Key Vault. Here are the key aspects of Azure Key Vault access policies:

- **Permissions and operations:**

- Access policies determine the operations that can be performed on Key Vault resources, such as reading, writing, deleting, and managing keys, secrets, and certificates.
- Permissions can be granted at a granular level, allowing for fine-grained access control.
- Common permissions include get, list, set, update, delete, backup, restore, and more.

- **Role-based access control (RBAC):**

- Azure Key Vault leverages Azure RBAC to assign access policies.
- RBAC enables you to assign roles to users, groups, or applications, which determine the permissions they have within Key Vault.
- Built-in roles include Key Vault Administrator, Key Vault Contributor, and Key Vault Reader, each with different levels of access.

- **Principle of least privilege:**

- Access policies follow the principle of least privilege, granting only the necessary permissions to perform specific operations.

- By assigning permissions on a need-to-know basis, you can minimize the risk of unauthorized access and potential misuse of Key Vault resources.
- **Multiple Access Policies:**
 - Azure Key Vault allows you to define multiple access policies, each with its own set of permissions.
 - This enables you to have different levels of access for different users, applications, or groups.
 - Multiple access policies can be useful when you need to grant different permissions to different entities within your organization.
- **Access policy management:**
 - Access policies can be managed and modified through the Azure portal, Azure CLI, Azure PowerShell, or Azure SDKs.
 - You can add, remove, or modify access policies as needed, ensuring that access rights are always up to date and aligned with your security requirements.
- **Azure AD integration:**
 - Azure Key Vault integrates with **Azure Active Directory (Azure AD)** for authentication and authorization.
 - Access policies are tied to Azure AD identities, allowing you to grant access to specific users, groups, or applications based on their Azure AD membership.

By effectively managing access policies in Azure Key Vault, you can control and secure access to your cryptographic keys, secrets, and certificates. By adhering to the principle of least privilege and assigning permissions based on specific roles and responsibilities, you can ensure that only authorized entities have the necessary access to Key Vault resources.

Case study

In this case study, we will walk through the process of creating a Web Application and securely retrieving a secret from Azure Key Vault. Azure Key Vault plays a pivotal role in ensuring the confidentiality of sensitive information, such as passwords, API keys, and connection strings, used by our application.

By leveraging Azure Key Vault's secure storage and fine-grained access control, we will demonstrate how to protect these critical secrets from exposure within our code or configuration files. Instead, our Web Application will authenticate with Azure Key Vault and request the required secret when needed, following the principle of least privilege.

Throughout the case study, we will showcase the step-by-step implementation of this approach, emphasizing the best practices for securing secrets within Azure Key Vault. By the end of this study, you will gain the knowledge and practical skills to safeguard your web applications' sensitive information using Azure Key Vault, enhancing the overall security of your application infrastructure.

Creating Azure Key Vault

Creating an Azure Key Vault involves several steps. In the following section, the high-level process to create an Azure Key Vault using the Azure portal is outlined:

1. **Sign in to the Azure Portal:** Go to <https://portal.azure.com> and sign in with your Azure account.
2. **Create a new resource:** Click on the "+" **Create a resource** button on the left-hand side of the Azure portal.
3. **Search for "Key Vault":** In the search bar, type "Key Vault" and select "Key Vault" from the list of results.
4. **Click "Create":** On the Key Vault overview page, click the "Create" button to start the creation process.
5. Fill in the basic information:
 - **Subscription:** Choose the appropriate Azure subscription you want to use for the Key Vault.
 - **Resource group:** Create a new resource group or select an existing one. A resource group is a logical container for resources in Azure.
 - **Key vault name:** Enter a unique name for your Key Vault. The name must be globally unique across all Azure Key Vault instances.
 - **Region:** Select the Azure region where you want your Key Vault to be located. Choose the region closest to

your application for lower latency.

10. **Configure access policy (optional):** You have the option to configure access policies during creation or later. Access policies control who can access the Key Vault and perform operations on its resources. For now, you can skip this step and set up access policies later.
11. **Configure advanced settings (optional):** You can set up advanced configurations like enabling soft-delete, purge protection, and network access control during creation if needed. Otherwise, you can use the default settings.
12. **Review + create:** Review the configuration details and click the "Create" button to start the deployment.
13. **Wait for deployment:** The Key Vault creation process may take a few moments. Wait for the deployment to complete.
14. **Access your Key Vault:** Once the Key Vault is created successfully, you can access it from the Azure portal. Navigate to "All resources" or search for the Key Vault's name to find it.

That is it! You have now created an Azure Key Vault. After creating the Key Vault, you can start storing your cryptographic keys, secrets, and certificates securely, and you can further configure access policies to control who can access them and what operations they can perform. You can see the overview page of the Azure Key Vault below:

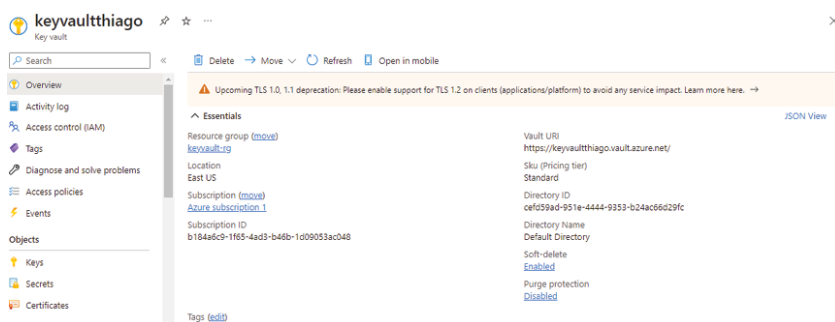


Figure 7.1: Azure Key vault recently created.

Managing Key Vault Access policies

To ensure secure management of the secrets within the Azure Key Vault, you should configure the access policies to grant appropriate permissions to authorized users. Follow the steps below to give permissions to manage the secrets to your user:

1. **Navigate to the Azure Key Vault:** Sign in to the Azure portal and locate your Azure Key Vault by either searching for its name or accessing it from the "**All resources**" menu.
2. **Enter Access policies:** Inside the Key Vault, select the "**Access policies**" tab from the left-hand navigation menu.
3. **Add an Access policy:** Click on the "**Add Access policy**" button to define a new access policy for your user.
4. **Define permissions:** In the "**Add Access policy**" panel, specify the necessary permissions you want to grant to your user. For managing secrets, ensure you select "**Get**" and "**Set**" permissions. "**Get**" allows reading secrets, while "**Set**" allows creating or updating secrets.
5. **Assign user or group:** Under "**Select principal**," search for and choose the user or group to which you want to grant these permissions. You can select from your Azure AD users or groups.
6. **Save the access policy:** Click "**Add**" to save the access policy and apply the permissions to the specified user or group.
7. **Verify access:** After saving the access policy, your user will now have the necessary permissions to manage the secrets within the Azure Key Vault.

It is essential to ensure that you grant permissions only to trusted individuals or groups and follow the principle of least privilege, giving users the minimum required permissions for their tasks. Regularly review and manage access policies to maintain a secure environment and protect your sensitive data effectively. You can see a picture of the access policies configured for my user below:

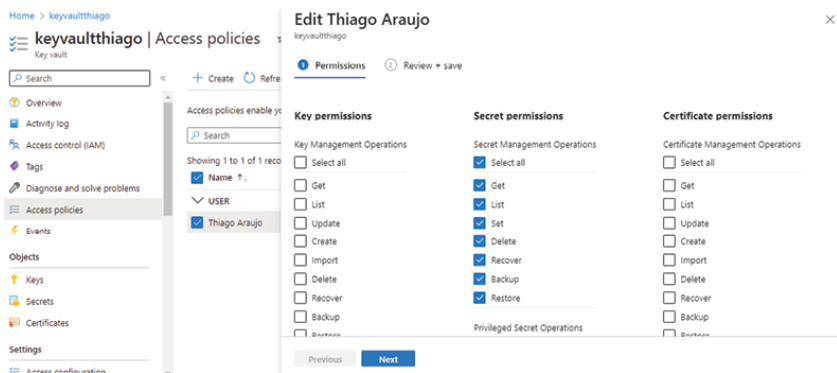


Figure 7.2: Azure Key Vault policies for my user.

Accessing a key

To kickstart our project, we will create a web application and then proceed to install the required NuGet packages listed below:

- Azure.Security.KeyVault.Secrets
- Azure.Identity

We can see the solution of our recently created project with the nuget packages in the image below:

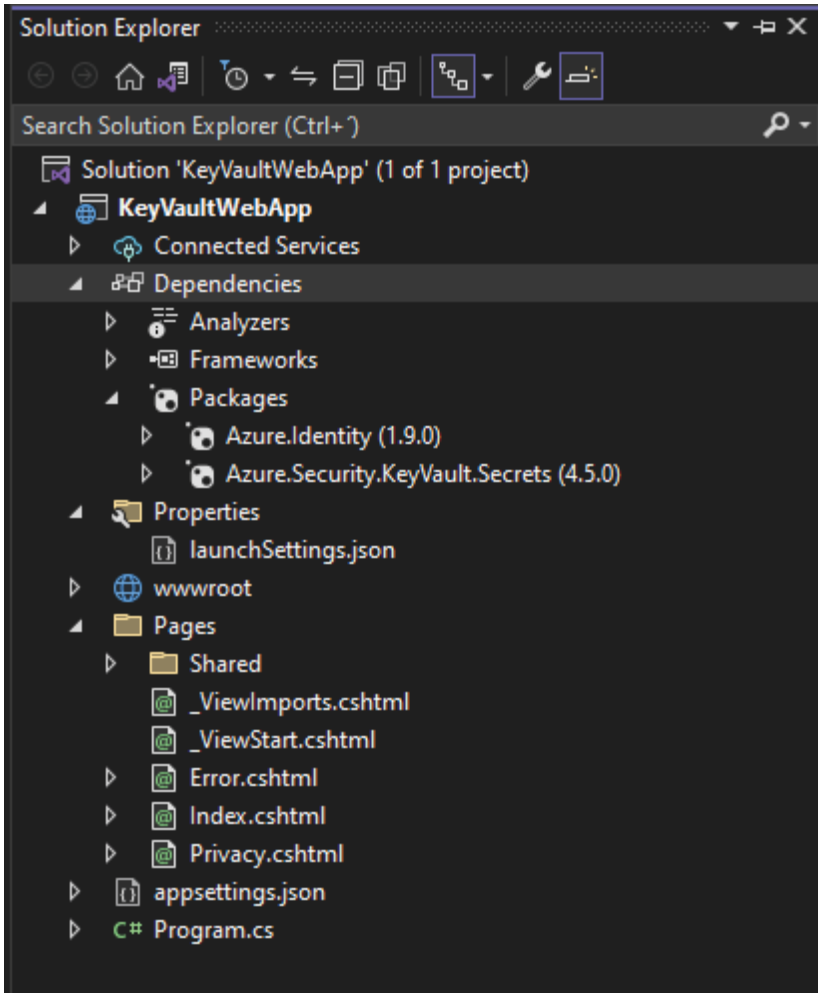


Figure 7.3: Web application project solution.

To ensure seamless integration and access to Azure Key Vault from Visual Studio, it is essential to configure Azure Service Authentication with the same Azure account used to create the Azure Key Vault. Follow these steps to verify and set up the configuration:

1. **Verify Azure Account:** Confirm that you are signed in to Visual Studio with the correct Azure account that was used to create the Azure Key Vault.
2. **Open Visual Studio:** Launch Visual Studio and go to the

"Tools" menu.

3. **Options:** From the "Tools" menu, select "Options."
4. **Azure Service Authentication:** In the "Options" window, expand "Azure Service Authentication" under the "Azure" category.
5. **Choose account:** Ensure that the dropdown menu for "Account Selection" displays the same Azure account associated with the Azure Key Vault.
6. **Verify credentials:** If needed, click "Manage Account" to verify the account's credentials or add the correct account if it's not listed.
7. **Authenticate:** If prompted, authenticate with the Azure account to ensure the credentials are up-to-date and valid.

By confirming that your Visual Studio's Azure Service Authentication is correctly configured with the same Azure account as the one used to create the Azure Key Vault, you will enable smooth communication between your application and the Key Vault. This alignment ensures that your Web Application can securely retrieve and manage secrets from the Azure Key Vault without any authentication issues, as we can see the account authenticated from the picture below:

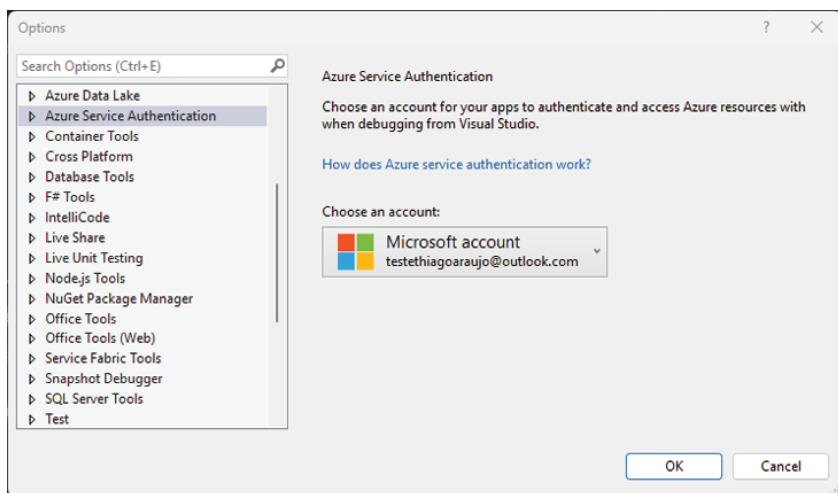


Figure 7.4: Visual Studio Azure Service Authentication with my account.

Now, we have to create a secret and to create a secret in your Azure Key Vault securely, follow these steps:

1. **Navigate to Key Vault:** Sign in to the Azure portal (<https://portal.azure.com>) and navigate to your Azure Key Vault by searching for its name or finding it in the "All resources" list.
2. **Access Secrets:** Inside the Key Vault, select the "Secrets" option from the left-hand navigation menu.
3. **Add a New Secret:** Click on the "+ Generate/Import" button to create a new secret.
4. **Define Secret Details:** In the "Create a secret" panel, provide the necessary information:
 - **Name:** Enter a unique name for the secret.
 - **Value:** Enter the actual secret value you want to store securely.
 - **Content type (optional):** You can specify the content type if needed, for example, "text/plain" or "application/json".
8. **Set Activation Date and Expiration (optional):** If required, you can set the activation date and expiration time for the secret. This is useful for managing the secret's validity period.
9. **Click "Create":** After filling in the details, click the "Create" button to add the secret to your Key Vault.

The secret is now securely stored in your Azure Key Vault, protected by Azure's robust security measures. By utilizing the Key Vault for secret management, you ensure sensitive information, such as connection strings, API keys, and passwords, remains safe and inaccessible to unauthorized users. Furthermore, your Web Application can now retrieve this secret securely using Azure Key Vault APIs, ensuring the confidentiality and integrity of your sensitive data. We can see our recently created secret from the picture below:



14ca0916977b4177bf25ca4cfdaddaca



Secret Version

Updated

7/29/2023, 2:32:19 PM

Secret Identifier

<https://keyvaultthiago.vault.azure.net/secrets/thiagosecret>

Settings

Set activation date ⓘ

☐

Set expiration date ⓘ

☐

Enabled

Yes No

Tags

0 tags

Secret

Content type (optional)

Show Secret Value

Secret value

.....



Figure 7.5: A new secret in Azure Key Vault.

To retrieve our secret from the Web Application, we must change the **Index.cshtml.cs** as follows:

```
1. public class IndexModel : PageModel
2. {
3.     private readonly
   ILogger<IndexModel> _logger;
4.     public string Message { get; set; }
5.
6.     public
   IndexModel(ILogger<IndexModel> logger)
7.     {
8.         _logger = logger;
9.     }
10.
11.    public void OnGet ()
12.    {
13.        try
```

```

14.         {
15.             SecretClientOptions options
= new SecretClientOptions()
16.         {
17.             Retry =
18.             {
19.                 Delay=
20.                 TimeSpan.FromSeconds(2),
21.                 MaxDelay =
22.                 TimeSpan.FromSeconds(16),
23.                 MaxRetries = 5,
24.                 Mode =
25.                 RetryMode.Exponential
26.             }
27.         };
28.
29.         var client = new
30.         SecretClient(new Uri("https://
31.         keyvaultthiago.vault.azure.net/"), new
32.         DefaultAzureCredential(), options);
33.
34.         KeyVaultSecret secret =
35.         client.GetSecret("thiogosecret");
36.
37.         Message = secret.Value;
38.     }
39. }

```

To display our secret message, we must update the **Index.cshtml** as follows:

```


```

```

1. @page
2. @model IndexModel
3. @{
4.     ViewData["Title"] = "Home page";
5. }
6.
7. <div class="text-center">
8.     <h1 class="display-4">Welcome</h1>
9.     <p>This is the secret message:
        @Model.Message</p>
10. </div>

```

The following will be the successful output:

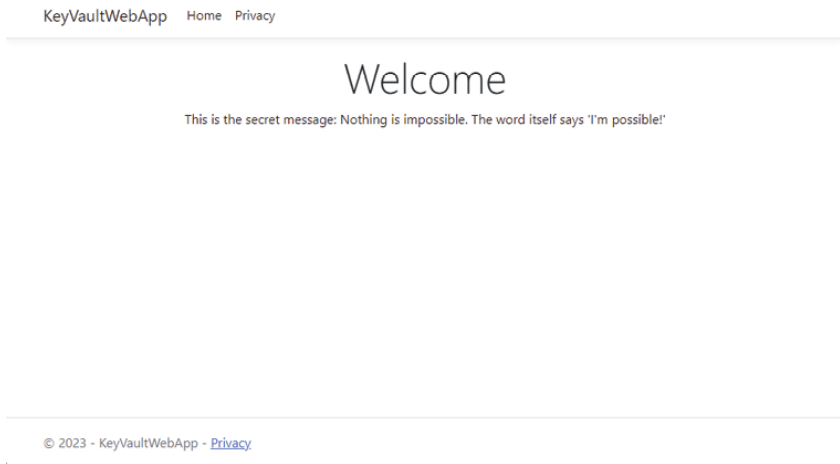


Figure 7.6: Web application displaying the secret retrieved from Azure Key Vault.

Conclusion

In this chapter, we explored the importance of securing your applications with Azure Key Vault and discussed various aspects of its implementation. We began with an overview of Azure Key Vault, understanding its capabilities and benefits in ensuring the security of cryptographic keys, secrets, and certificates.

Authentication emerged as a crucial factor in securing Azure Key

Vault, and we delved into different authentication mechanisms available, learning how to configure and manage them effectively. By implementing strong authentication measures, you can ensure that only authorized entities can access your sensitive data.

Access policies played a significant role in controlling access to Azure Key Vault resources, and we examined how to define and manage granular permissions through access policies. By carefully managing these policies, you can maintain a strong security posture and minimize the risk of unauthorized access.

To provide a practical perspective, we explored a case study that demonstrated the step-by-step implementation of Azure Key Vault in a real-world scenario. From creating and configuring the key vault to defining and managing access policies, the case study exemplified the practical application of Azure Key Vault for securing your applications.

Additionally, we showcased an example of accessing a key stored in Azure Key Vault through a .NET console application. This demonstrated the integration of your applications with Azure Key Vault, enabling secure retrieval and utilization of keys.

By following the knowledge and skills presented in this chapter, you now possess the tools and understanding to secure your applications effectively using Azure Key Vault. You have learned the importance of robust authentication, the significance of access policies, and how to leverage Azure Key Vault's features for enhanced application security.

Remember to adhere to best practices for securing your applications with Azure Key Vault, and regularly review and update your security measures to stay ahead of potential threats. By adopting Azure Key Vault as an integral part of your application security strategy, you can protect your valuable assets, ensure data confidentiality, and build trust among your users.

Congratulations on completing this chapter! You are now equipped with the knowledge and skills to fortify your applications and defend against potential security threats with Azure Key Vault. Embrace this newfound understanding and continue to prioritize the security of your applications in today's ever-evolving digital landscape.

The next chapter, *Building Dynamic Web Apps with Blazor and ASP.NET*, explores essential topics in the development of dynamic web applications. The overview sets the stage for discussions on key features like Hot Reload, security considerations, and the advantages of strongly-typed databinding. The chapter also compares Blazor and Razor, providing insights into their roles and use cases. A case study with a step-by-step implementation offers practical learning, while guidance on creating a Blazor project and testing the Hot Reload feature enhances developers' understanding. The chapter concludes with discussions on implementing authorization/authentication and leveraging strongly-typed databinding effectively.

CHAPTER 8

Building Dynamic Web Apps with Blazor and ASP.NET

Introduction

Welcome to the chapter on building dynamic web applications with Blazor and ASP.NET. In this chapter, we will explore the powerful features and techniques that enable the development of interactive and responsive web apps using these technologies.

Blazor is a modern web framework that brings the power of .NET to the client-side. With Blazor, developers can use C# and Razor syntax to build rich web applications, seamlessly integrating them with server-side logic.

Complementing Blazor, ASP.NET is a mature and robust web development framework that provides a solid foundation for building scalable and high-performance web applications. It offers a wide range of features and tools that simplify the development process and enhance the overall user experience.

Throughout this chapter, we will embark on a journey to explore

the core concepts and techniques required to build dynamic web apps with Blazor and ASP.NET. We will cover several key topics that are crucial for building dynamic web apps with Blazor and ASP.NET.

Structure

This chapter covers the following topics:

- Web Apps with Blazor and .NET
- Hot reload
- Security
- Data binding
- Blazor vs Razor
- Practical case study

Objectives

As the chapter concludes, you will grasp key aspects. Explore Hot Reload, a feature enabling real-time code changes for immediate results, enhancing development efficiency. Navigate security essentials for Blazor and ASP.NET web apps, focusing on authentication, data protection, and best practices. Dive into Data Binding, ensuring type safety and code maintainability. Differentiate Blazor and Razor, understanding their strengths for informed web app development decisions. The chapter closes with a practical case study, reinforcing concepts. This comprehensive understanding equips you to construct dynamic web apps with Blazor and ASP.NET, empowering the development of interactive, feature-rich applications.

Web Apps with Blazor and .NET

Blazor is a framework that enables the creation of interactive client-side web user interfaces using .NET. With Blazor, developers can build rich UIs using C# instead of JavaScript, allowing for a more familiar and productive development experience.

One of the key advantages of using Blazor is the ability to share app logic between the server-side and client-side, leveraging the power

of .NET across both environments. This allows for code reusability and consistency throughout the application.

Blazor renders the UI as HTML and CSS, ensuring wide browser support, including mobile browsers. It also integrates seamlessly with modern hosting platforms like Docker, enabling easy deployment and scalability.

By using .NET for client-side web development, developers can write code in C# instead of JavaScript. They can tap into the vast ecosystem of .NET libraries, benefit from the performance, reliability, and security of .NET, and work with popular development environments like Visual Studio, on Windows, or Visual Studio Code, on Windows, Linux, or macOS. The adoption of Blazor and .NET for client-side web development provides developers with a stable, feature-rich, and user-friendly set of languages, frameworks, and tools. It opens up new possibilities for building hybrid desktop and mobile apps, making it a compelling choice for modern web development projects.

Hot reload

With .NET hot reload, you can apply code changes, including modifications to stylesheets, to a running app without the need to restart it or lose the app's current state. This feature is available for all ASP.NET Core 6.0 and later projects.

When making code updates, the changes take effect in the running app under certain conditions. For instance, startup logic that runs only once, such as middleware (except for inline middleware delegates), configured services, and route creation and configuration, will be rerun to incorporate the changes. Inline middleware delegates are delegates used within the context of middleware functions in frameworks such as asp.net core that are entirely implemented inline within the middleware pipeline configuration. This means that language developers can define the logic which the middleware will execute from within the middleware configuration code without creating separate methods or classes for defining named middleware.

In Blazor apps, the Razor component render is automatically triggered by the framework. On the other hand, in MVC and Razor

Pages apps, hot reload will automatically refresh the browser. It's important to note that removing a Razor component parameter attribute does not cause the component to re-render. To reflect such changes, restarting the app is necessary.

The introduction of .NET hot reload significantly enhances the development experience by providing seamless and immediate code updates during the app's runtime. It empowers developers to iterate quickly and efficiently, eliminating the need for frequent restarts and allowing them to focus on building and refining their applications.

Supported frameworks and application types

The following table provides an overview of the application types, that offer support for .NET hot reload. It highlights whether hot reload is available when the debugger is attached (*F5*) or when running without the debugger (*Ctrl+F5*). Additionally, it indicates whether minimum support necessitates the use of .NET 6. It is worth noting that .NET 6 is always required for *Ctrl+F5* support. The table also includes the minimum version of Visual Studio that incorporates this hot reload feature.

Please refer to the following table for further details:

ASP.NET Core MVC			
ASP.NET Core Web API			
ASP.NET Core SignalR			
ASP.NET Core Identity			
ASP.NET Core Authentication			
ASP.NET Core Authorization			
ASP.NET Core Data Protection			
ASP.NET Core Logging			
ASP.NET Core Health Checks			
ASP.NET Core OpenAPI			
ASP.NET Core Performance Monitoring			
ASP.NET Core Tracing			
ASP.NET Core Metrics			
ASP.NET Core Configuration			
ASP.NET Core Options			
ASP.NET Core Environment			
ASP.NET Core Hosting			
ASP.NET Core Deployment			
ASP.NET Core Security			
ASP.NET Core Testing			
ASP.NET Core DevOps			
ASP.NET Core CI/CD			
ASP.NET Core Monitoring			
ASP.NET Core Alerting			
ASP.NET Core Dashboarding			
ASP.NET Core Reporting			
ASP.NET Core Analytics			
ASP.NET Core Compliance			
ASP.NET Core Accessibility			
ASP.NET Core Internationalization			
ASP.NET Core Localization			
ASP.NET Core Globalization			
ASP.NET Core Cryptography			
ASP.NET Core Networking			
ASP.NET Core Database			
ASP.NET Core Cache			
ASP.NET Core Queue			
ASP.NET Core Scheduler			
ASP.NET Core Workflow			
ASP.NET Core Business Logic			
ASP.NET Core Domain Logic			
ASP.NET Core Application Logic			
ASP.NET Core Presentation Logic			
ASP.NET Core Infrastructure Logic			
ASP.NET Core Integration Logic			
ASP.NET Core External Services			
ASP.NET Core Third-Party Libraries			
ASP.NET Core Open-Source Projects			
ASP.NET Core Commercial Products			
ASP.NET Core Consulting Services			
ASP.NET Core Training Courses			
ASP.NET Core Certification Programs			
ASP.NET Core Research Papers			
ASP.NET Core Industry Reports			
ASP.NET Core Market Analysis			
ASP.NET Core Future Trends			
ASP.NET Core Career Opportunities			
ASP.NET Core Salary Surveys			
ASP.NET Core Job Postings			
ASP.NET Core Recruitment Agencies			
ASP.NET Core Freelance Platforms			
ASP.NET Core Remote Work Opportunities			
ASP.NET Core Contract Positions			
ASP.NET Core Full-Time Positions			
ASP.NET Core Part-Time Positions			
ASP.NET Core Internship Programs			
ASP.NET Core Apprenticeship Programs			
ASP.NET Core Mentorship Programs			
ASP.NET Core Career Coaching			
ASP.NET Core Resume Writing			
ASP.NET Core Interview Preparation			
ASP.NET Core Job Search Strategies			
ASP.NET Core Networking Tips			
ASP.NET Core Industry Conferences			
ASP.NET Core Meetups			
ASP.NET Core Workshops			
ASP.NET Core Seminars			
ASP.NET Core Webinars			
ASP.NET Core Podcasts			
ASP.NET Core YouTube Channels			
ASP.NET Core Blogs			
ASP.NET Core Forums			
ASP.NET Core Social Media Groups			
ASP.NET Core LinkedIn Groups			
ASP.NET Core Facebook Groups			
ASP.NET Core Twitter Chats			
ASP.NET Core Reddit Communities			
ASP.NET Core Discord Servers			
ASP.NET Core Slack Channels			
ASP.NET Core Telegram Groups			
ASP.NET Core WhatsApp Groups			
ASP.NET Core Email Newsletters			
ASP.NET Core RSS Feeds			
ASP.NET Core Podcast Feeds			
ASP.NET Core Video Feeds			
ASP.NET Core Blog Feeds			
ASP.NET Core Forum Feeds			
ASP.NET Core Social Media Feeds			
ASP.NET Core Email Lists			
ASP.NET Core Mailing Lists			
ASP.NET Core Distribution Lists			
ASP.NET Core Contact Lists			
ASP.NET Core Lead Lists			
ASP.NET Core Customer Lists			
ASP.NET Core Prospect Lists			
ASP.NET Core Partner Lists			
ASP.NET Core Vendor Lists			
ASP.NET Core Supplier Lists			
ASP.NET Core Client Lists			
ASP.NET Core Stakeholder Lists			
ASP.NET Core Board Lists			
ASP.NET Core Advisory Lists			
ASP.NET Core Consultant Lists			
ASP.NET Core Freelancer Lists			
ASP.NET Core Contractor Lists			
ASP.NET Core Subcontractor Lists			
ASP.NET Core Vendor Lists			
ASP.NET Core Supplier Lists			
ASP.NET Core Client Lists			
ASP.NET Core Stakeholder Lists			
ASP.NET Core Board Lists			
ASP.NET Core Advisory Lists			
ASP.NET Core Consultant Lists			
ASP.NET Core Freelancer Lists			
ASP.NET Core Contractor Lists			
ASP.NET Core Subcontractor Lists			
ASP.NET Core Vendor Lists			
ASP.NET Core Supplier Lists			
ASP.NET Core Client Lists			
ASP.NET Core Stakeholder Lists			
ASP.NET Core Board Lists			
ASP.NET Core Advisory Lists			
ASP.NET Core Consultant Lists			
ASP.NET Core Freelancer Lists			
ASP.NET Core Contractor Lists			
ASP.NET Core Subcontractor Lists			
ASP.NET Core Vendor Lists			
ASP.NET Core Supplier Lists			
ASP.NET Core Client Lists			
ASP.NET Core Stakeholder Lists			
ASP.NET Core Board Lists			
ASP.NET Core Advisory Lists			
ASP.NET Core Consultant Lists			
ASP.NET Core Freelancer Lists			
ASP.NET Core Contractor Lists			
ASP.NET Core Subcontractor Lists			
ASP.NET Core Vendor Lists			
ASP.NET Core Supplier Lists			
ASP.NET Core Client Lists			
ASP.NET Core Stakeholder Lists			
ASP.NET Core Board Lists			

Table 8.1: Hot reload support

When coding in C#, we can leverage the hot reload feature to make code changes and experience the benefits of the .NET hot reload. This experience is made possible by the Edit and Continue

mechanism, which introduces several enhancements. These improvements expand the scope of supported edits, surpassing the limitations of previous versions of Visual Studio. Notable improvements include:

- Adding, updating, or deleting Custom Attributes
- Adding or updating Record structs
- Adding or updating `#line` directives
- Editing Switch expressions
- Editing files with `#line` directives, including changes to the directive itself
- Editing top-level statements
- Editing code that uses any of the new C# 10 features, such as global using directives, file scoped namespaces, improved lambdas, and parameter-less struct constructors
- Renaming Lambda parameters
- Renaming parameters of existing methods

Unsupported Scenarios

While hot reload provides a convenient and efficient method for updating code on-the-fly during development, it is important to note that there are certain scenarios where hot reload may not be supported. These unsupported scenarios can arise due to various factors, such as the nature of code changes, specific runtime conditions, or limitations in the development environment. It is crucial for developers to be aware of these scenarios to ensure a smooth and effective coding experience. By understanding the limitations of hot reload, developers can make informed decisions and utilize alternative approaches when necessary. You can find the most common scenarios where Hot-Reload is not supported, mentioned as follows:

- Xamarin.Forms in iOS and Android scenarios, with partial support for a UWP app.
- If the Edit and Continue settings are disabled in your Visual Studio.
- If PublishTrimmed is set to True in your debug profile.
- If PublishReadyToRun is set to True in your debug profile.

- WinUI 3 apps with the property `nativeDebugging` not set, or set to true in your `LaunchSettings.json` file.

Configuring Hot Reload

Setting up hot reload for your development environment is a simple and swift process. With just a few straightforward steps, you can enable hot reload and start leveraging its benefits in your coding workflow. hot reload seamlessly integrates with popular development tools and frameworks, making it easy to incorporate into your existing setup. By following the straightforward setup instructions, you will be able to quickly take advantage of the fast and efficient hot reload feature, enhancing your development experience and accelerating your coding iterations.

To enable and configure hot reload go to **Tools | Options | Debugging | .NET/C++ hot reload**, refer to the following figure:

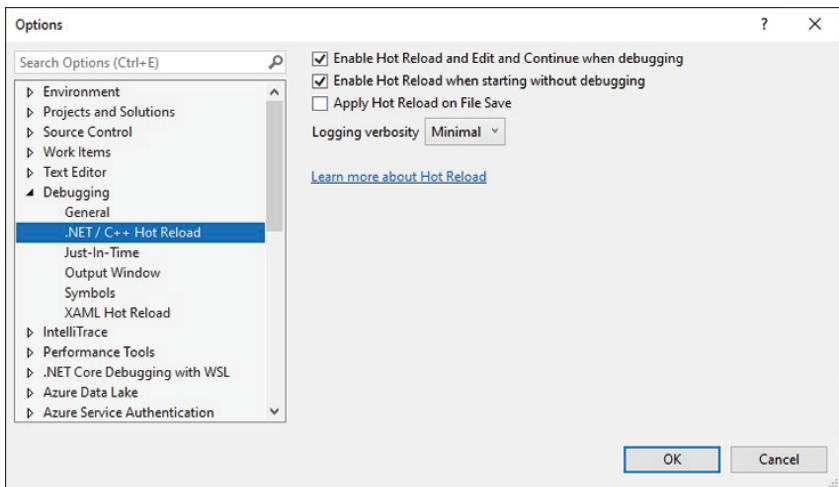


Figure 8.1: Hot reload configuration window.

As mentioned before, the process of enabling hot reload is straightforward. The settings for hot reload include:

- Enable hot reload and Edit and Continue when debugging.
 - This setting enables hot reload when starting the application with the debugger attached (`F5`), allowing for instant code updates during the debugging process.
- Enable hot reload when starting without debugging.

- This setting enables hot reload when starting the application without the debugger attached (*Ctrl+F5*), enabling code updates in real-time even when not actively debugging.
- Apply hot reload on File Save.
- This setting automatically applies any code changes made when you save the file, ensuring that the updates take effect immediately.

Security

Security considerations vary between Blazor Server and Blazor WebAssembly apps. In Blazor Server apps, running on the server-side, authorization checks have the capability to determine the UI options available to a user, such as menu entries, as well as access rules for different app areas and components.

On the other hand, Blazor WebAssembly apps run on the client-side, and authorization is primarily used to determine which UI options to display. Since client-side checks can be modified or bypassed by users, Blazor WebAssembly apps are unable to enforce authorization access rules.

When it comes to authorization conventions in Razor Pages, they do not directly apply to routable Razor components. However, if a non-routable Razor component is embedded within a page of a Razor Pages app, the page's authorization conventions indirectly impact the Razor component along with the rest of the page's content.

ASP.NET Core Identity, designed for HTTP request and response communication, does not align perfectly with the Blazor app client-server communication model. Therefore, it is recommended that ASP.NET Core apps utilizing ASP.NET Core Identity for user management opt for Razor Pages instead of Razor components for Identity-related UI tasks such as user registration, login, logout, and user management. While it is possible to build Razor components that handle Identity tasks directly in certain scenarios, Microsoft does not endorse or provide support for this approach.

It's important to note that ASP.NET Core abstractions like **SignInManager<TUser>** and **userManager<TUser>** are not supported within Razor components. For detailed guidance on

utilizing ASP.NET Core Identity with Blazor, you can refer to the "Scaffold ASP.NET Core Identity into a Blazor Server app" resource through the following link <https://learn.microsoft.com/en-us/aspnet/core/security/authentication/scaffold-identity>.

Considering these security aspects and following recommended practices will help ensure the effective implementation of authorization and authentication mechanisms within your Blazor applications.

Blazor leverages the authentication mechanisms already present in ASP.NET Core to authenticate the user's identity. The specific mechanism employed depends on the hosting model of the Blazor application, whether it is Blazor Server or Blazor WebAssembly.

Blazor server authentication

Blazor Server operates over a SignalR connection with the client. Authentication in SignalR-based apps is handled when the connection is established. Authentication can be based on a cookie or some other bearer token, but authentication is managed entirely within the circuit via the SignalR hub.

The built-in **AuthenticationStateProvider** service for Blazor Server apps obtains authentication state data from ASP.NET Core's **HttpContext.User**. This is how the authentication state integrates with existing ASP.NET Core authentication mechanisms.

Blazor WebAssembly authentication

In Blazor WebAssembly, it is important to acknowledge that authentication checks can potentially be bypassed since all client-side code is accessible and modifiable by users. This characteristic applies not only to Blazor WebAssembly but also to other client-side app technologies, including JavaScript **Single Page Application (SPA)** frameworks and native apps across different operating systems. Developers should be aware of this inherent nature when implementing authentication mechanisms in Blazor WebAssembly applications.

To handle authentication in Blazor WebAssembly, we must make usage of the built-in or custom **AuthenticationStateProvider** service. The **AuthenticationStateProvider** serves as the

underlying service utilized by the **AuthorizeView** component and **CascadingAuthenticationState** component to retrieve the authentication state for a user.

Typically, you would not directly interact with the **AuthenticationStateProvider**. Instead, it is recommended to utilize the **AuthorizeView** component or the **Task<AuthenticationState>**. The advantage of using these approaches is that they handle the automatic notification of any changes in the underlying authentication state data.

Directly using the **AuthenticationStateProvider** can have a drawback where the component is not automatically notified when there are changes in the authentication state data. Therefore, leveraging the higher-level components or approaches provided is generally preferred for seamless and reliable authentication state management.

Authorization

The authorization verification starts exactly after a user is successfully authenticated, if a user cannot authenticate then the authorization verification is not needed. Authorization is the process of controlling access to specific components, pages, or actions within a Blazor application based on the authenticated user's identity and assigned permissions. It involves implementing security measures, such as authentication, to verify the user's identity, and authorization, which determines whether the user possesses the necessary rights to perform certain operations or view a specific content.

Blazor provides built-in authorization features, including the **AuthorizeView** component and the **Authorize** attribute, which allow developers to easily apply authorization rules and restrict access based on roles or policies. By leveraging these features, developers can ensure that only authorized users can interact with sensitive features or view confidential information within a Blazor application, ensuring that only authenticated users with the appropriate role, claim, or policy fulfillment can access restricted functionality and information.

AuthorizeView component

The **AuthorizeView** component in Blazor offers the ability to selectively show or hide UI content based on the user's authorization status. This feature proves beneficial when there is a need to display user-specific data without utilizing the user's identity in procedural logic.

By utilizing the **AuthorizeView** component in a Razor page, developers can access the context variable of type **AuthenticationState** (`@context` in Razor syntax), which provides access to essential information about the currently signed-in user.

Authorize attribute

In Razor components, you can utilize the `[Authorize]` attribute for authorization purposes. This attribute supports both role-based and policy-based authorization. To implement role-based authorization, you can specify the `Roles` parameter, whereas the `Policy` parameter is used for policy-based authorization.

If neither roles nor policy is specified, the `[Authorize]` attribute applies the default policy, where authenticated users are authorized and unauthenticated users are unauthorized. In cases where the user is not authorized and the app does not customize unauthorized content using the Router component, the framework automatically displays the fallback message **"Not Authorized."**

Data binding

Razor components offer convenient data binding capabilities through the use of the `@bind` Razor directive attribute, which can be applied to fields, properties, or Razor expressions. It's important to note that the UI is updated when the component is rendered, rather than immediately upon modifying the field or property value. Typically, field and property updates are reflected in the UI after the execution of event handler code, as components render themselves following event triggers.

To bind a property or field to other **Document Object Model (DOM)** events, you can include an `@bind:event="{EVENT}"` attribute, replacing `{EVENT}` with the desired DOM event. Additionally, you can use the `@bind:after="{EVENT}"` attribute with a DOM event placeholder to execute asynchronous logic after

the binding process. It's worth mentioning that assigned C# methods are only executed when the bound value is synchronously assigned.

For two-way data binding, components support the definition of a pair of parameters: `@bind:get` and `@bind:set`. The `@bind:get` specifies the value to bind, while the `@bind:set` defines a callback for handling value changes. It's important to note that the `@bind:get` and `@bind:set` modifiers should always be used together.

Please note that using an event callback parameter with `@bind:set` (e.g., `[Parameter] public EventCallback<string> ValueChanged { get; set; }`) is not supported. Instead, it is recommended to pass a method that returns an Action or Task to the `@bind:set`.

Lastly, it is crucial to remember that Razor attribute binding is case-sensitive. Valid attributes include `@bind`, `@bind:event`, and `@bind:after`. Any variations with capital letters, such as `@Bind/` `@bind:Event/` `@bind:after` or `@BIND/` `@BIND:EVENT/` `@BIND:AFTER`, are considered invalid.

Two-way data binding

Blazor does not automatically synchronize the values between **Document Object Model (DOM)** elements and .NET variables unless they are bound using the `@bind` syntax. This is because Blazor is unaware of the intention to modify the .NET variable value within the event handler. In previous versions of Blazor, two-way data binding was accomplished by binding the element to a property and controlling the property's value through its setter.

For effective two-way data binding in Blazor, it is recommended to use the `@bind:get` and `@bind:set` modifiers instead of event handlers. Two-way data binding cannot be achieved solely through an event handler. By utilizing the `@bind:get` and `@bind:set` modifiers, you can establish two-way data binding seamlessly.

By using the `@bind:get` and `@bind:set` modifiers, you can not only manage the underlying value of the .NET variable through `@bind:set` but also bind the value of the .NET variable to the element's value through `@bind:get`. This comprehensive approach

ensures effective two-way data binding in your Blazor application.

Chained data binding

In scenarios where you need to bind a property of a child component to a property in its parent component, a common practice is to establish a chained bind. This involves simultaneous binding at multiple levels. To achieve this, you can utilize the `@bind-{PROPERTY}` syntax in the parent component's component parameters. Simply replace the `{PROPERTY}` placeholder with the desired property name that you want to bind.

However, it is important to note that chained binds cannot be implemented directly using the `@bind` syntax within the child component. Instead, you need to define an event handler and a separate value to facilitate the updating of the parent's property from the child component.

Nevertheless, the parent component can still employ the `@bind` syntax to set up the data binding with the child component.

As a naming convention, the `EventCallback<TValue>` associated with the parameter should be named using the component parameter name followed by the "Changed" suffix. The naming convention follows the syntax `{PARAMETER NAME}Changed`, where `{PARAMETER NAME}` represents the actual parameter name.

To trigger the delegate associated with the binding and notify the changes, you can utilize `EventCallback.InvokeAsync`, which invokes the delegate with the provided argument and dispatches an event notification for the updated property.

Blazor vs Razor

Comparing blazor to razor is a similar comparison like the comparison of Java to Javascript, they are totally different despite the similarity in their naming.

Blazor is a powerful framework that combines the flexibility of Razor components with the capability to run C# code and leverage the Razor view engine directly in the browser. Unlike Razor, which primarily focuses on server-based architecture and server-side templating, Blazor extends the functionality by enabling client-side logic using C# instead of JavaScript.

Razor components serve as the fundamental building blocks of any Blazor application, combining markup and code into cohesive units. These components are implemented with a `.razor` extension and allow developers to create dynamic user interfaces using the Razor syntax. With Blazor as the client-side hosting model, you can leverage the full potential of Razor components on the client-side.

Best practices

Blazor development is best approached by following certain recommended practices to ensure efficient, maintainable, and high-quality applications. These best practices encompass various aspects of Blazor development, including code organization, performance optimization, security, and user experience. By adhering to these best practices, developers can enhance the reliability, scalability, and overall success of their Blazor applications. The best practices are listed below, grouped by categories:

- Blazor Server

- Avoid using `IHttpContextAccessor` or `HttpContext`.

- The presence of `HttpContext` cannot be guaranteed within the `IHttpContextAccessor`, and even if `HttpContext` is accessible, it does not necessarily hold the context that initiated the Blazor application.

- To pass the request state to the Blazor application, the recommended approach is to utilize root component parameters during the initial rendering of the app. This allows for seamless integration and easy access to the required data. Alternatively, the application can also copy the relevant data into a scoped service during the initialization lifecycle event of the root component. This ensures the availability of the data throughout the application for efficient and consistent usage

- Avoid singleton services to share state

- It is important to exercise caution when passing request state in Blazor applications, as it can potentially introduce security vulnerabilities. For instance, in Blazor Server apps, where multiple app sessions coexist within the same server process, there is a risk of leaking user state across circuits. This is due to the fact that Blazor Server apps reside in the server's memory.

■ If designed appropriately, stateful singleton services can be utilized in Blazor apps. By following specific design considerations, these services can effectively maintain their state across multiple requests and provide consistent functionality throughout the Blazor application.

- The `authorize` attribute
 - It is recommended to utilize the attribute exclusively on `@page` components that are accessed through the Blazor Router.
 - To control the authorization of specific sections within a page, the `AuthorizeView` component should be used instead. By adhering to this practice, developers can effectively manage and enforce authorization rules within their Blazor applications.
- Two-way binding
 - When implementing two-way binding to a property with get/set accessors, it is necessary to discard the Task returned by `EventCallback.InvokeAsync`.
 - For achieving effective two-way data binding, it is highly recommended to utilize the `@bind:get` and `@bind:set` modifiers.

Practical case study

In this practical case study, we will delve into the world of Blazor Server App development. We will cover key topics such as creating a Blazor Server App project, harnessing the power of hot reload for seamless code updates, implementing authorization and authentication, and mastering data binding.

We will begin by creating a Blazor Server App project, understanding its structure and configuration. Then, we will explore the hot reload feature, which allows instant code updates during development, making the development process faster and more efficient.

Next, we will dive into authorization and authentication in Blazor Server Apps. We will explore how to establish user identity, enforce role-based access, and validate user claims for a secure application.

Lastly, we will uncover the power of data binding in Blazor Server. We will learn how to establish dynamic connections between data and the user interface, enabling automatic updates and synchronization.

Join us in this practical case study to gain hands-on experience and best practices for creating robust and interactive web applications with Blazor Server.

Creating a Blazor Server App project

In this section, we will explore the process of creating a Blazor Server App project. We will walk through the necessary steps to set up a Blazor Server application, including project creation, understanding the project structure, and configuring essential settings. By the end of this section, you will have a solid foundation to begin your Blazor Server development journey. Let us dive in and get started with building powerful web applications using Blazor Server!

The first step is to add a project of type Blazor Server App, as you can see in the following figure:

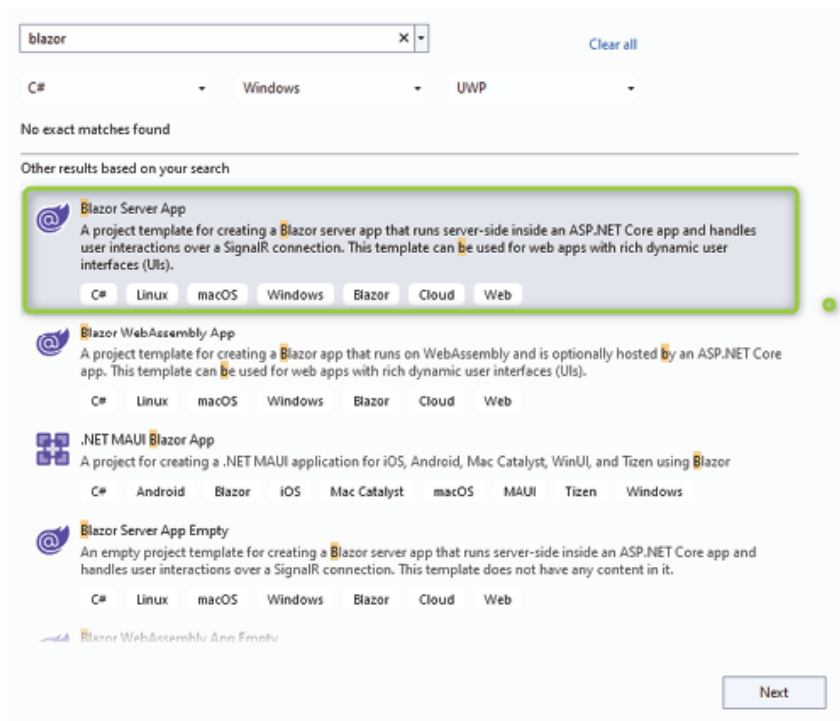


Figure 8.2: Creating a new project of type Blazor Server App

We have named my project **SampleBlazorApp** as Visual Studio created all those files.

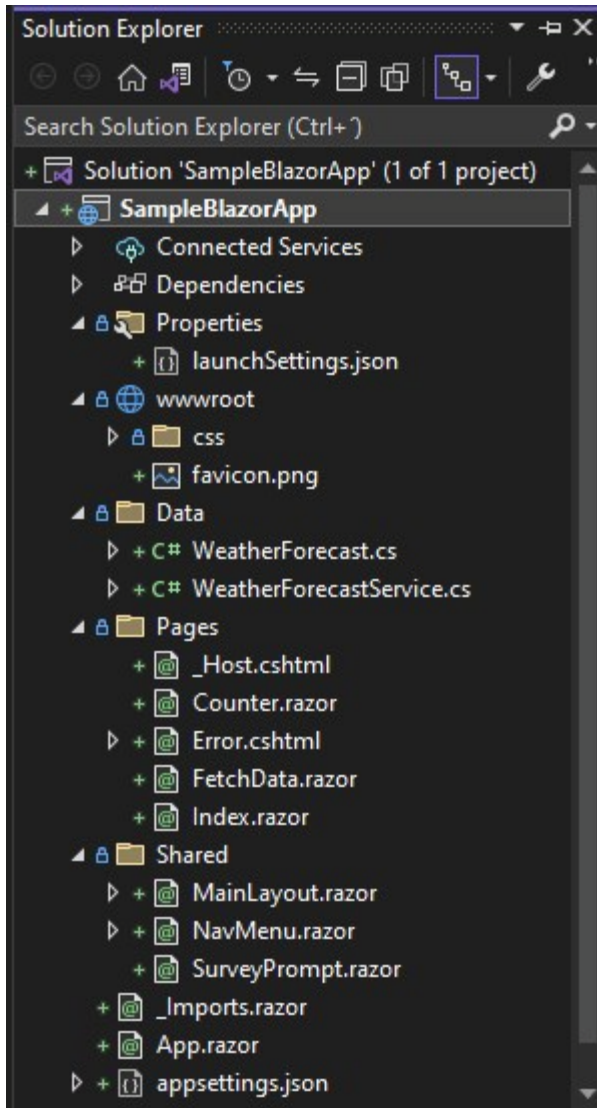


Figure 8.3: The Blazor Server App Solution Explore

Let us understand the purposes of each of these files:

- **launchSettings.json**

- It is a configuration file that plays a crucial role in debugging applications. It acts as a storage space for important configuration information, allowing you to define various settings for running your application

smoothly. This includes specifying the desired environment in which your application should operate, ensuring a seamless debugging experience. With **launchSettings.json**, you have the flexibility to customize and fine-tune your application's behavior during the debugging process.

- **wwwroot:**

- The Web Root folder is the designated location within your application where all the public static assets are stored. It serves as the central repository for files such as HTML, CSS, JavaScript, images, and other resources that are directly accessible to the clients. By organizing your app's static assets in the Web Root folder, you can ensure easy retrieval and efficient delivery of these files to enhance the user experience of your web application.

- **Data folder:** It contains the data and services to be presented in the Application.

- **WeatherForecast.cs**

- The class model used by the **WeatherForecastService**.

- This class model is used in the **FetchData.razor** component.

- **WeatherForecastService.cs**

- The service is used to show data in the **FetchData.razor** component.

- This class has a Singleton dependency injection defined in the **Program.cs**

- **Pages folder:** Contains the routable components/pages (**.razor**) that form the app, with each page's route specified using the **@page** directive.

- **_Host.cshtml**

- This page is rendered every time a user requests any page of the app.

- Defines the location where the root App component (**App.razor**) is rendered.

- This page is defined in the **Program.cs** class, in the **MapFallbackToPage** method.

- **Counter.razor**
 - The razor counter component is used on the counter page.
- **Error.cshtml** and **Error.cshtml.cs**
 - Default Error page, the user is redirected to this page when an unhandled exception occurs.
- **FetchData.razor**
 - Razor component used in the FetchData page.
 - Uses the **WeatherForecastService** to manage data.
- **Index.razor**
 - The **Index** component and Home page.
- **Shared folder:** Contains components and stylesheets shared among other pages.
- **MainLayout.razor**
 - Razor **MainLayout** component and the default layout for the app.
 - It is defined in the **App.razor**.
- **NavMenu.razor**
 - Razor **NavMenu** component and the navigation menu of the app.
 - The **MainLayout** component renders the NavMenu.
- **SurveyPrompt.razor**
 - Razor **SurveyPrompt** component.
 - It is called by the **Index** component.
- **_Imports.razor:**
 - The Imports file consists of common Razor directives that are typically included in the app's components (**.razor**), such as **@using** directives for namespaces. It allows you to conveniently import and reference necessary namespaces in your components without repetitive declarations.
- **App.razor:**

- The root component of the app.
- Responsible for establishing client-side routing using the Router component.
- It is called from the `_Host` page.
- **appsettings.json:**
 - Application level settings.
- **Program.cs:**
 - The application entry's point.
 - Has all the settings for the App startup and its pipelines.
 - Contains the Dependency Injection configuration.
 - Every configuration at the project level must be set here.

If we push *F5* or *CTRL + F5* we can see our app running in the browser:

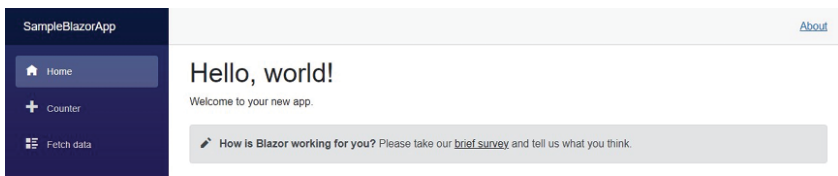


Figure 8.4: The Blazor Server App running in a browser.

Also, a command prompt will be launched with your app settings.

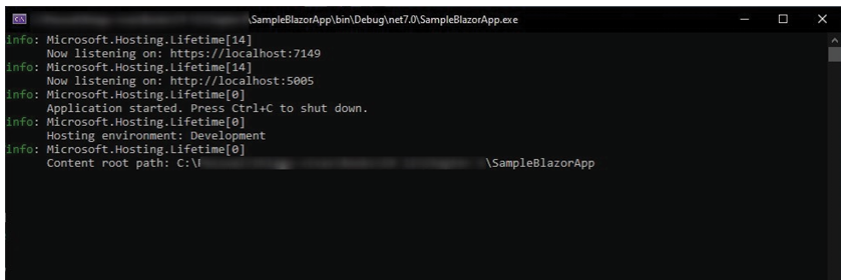


Figure 8.5: A Command Prompt launched by Visual Studio with the Application settings.

Working of hot reload

First, we are enabling the hot reload, like explained before. Then,

we are testing to make changes, while the project is running in debug mode (F5), in a **Razor** component and in the C# Weather Forecast Service.

With the project running in debug mode, the **Counter** component was updated as we can see below.

Counter component before updated:

```
1. @page "/"counter"
2. @attribute [Authorize]
3.
4. <PageTitle>Counter</PageTitle>
5.
6. <h1>Counter</h1>
7.
8. <p role="status">Current count:
  @currentCount</p>
9.
10. <button class="btn btn-primary"
    @onclick = "IncrementCount">Click me</button>
11.
12. @code {
13.     private int currentCount = 0;
14.
15.     private void IncrementCount ()
16.     {
17.         currentCount ++;
18.     }
19. }
```

Counter components after updated:

```
1. @page "/"counter"
2.
3. <PageTitle>Counter</PageTitle>
4.
5. <h1>Counter</h1>
6.
7. <p role="status">Current count:
```

```

@currentCount</p>
8.
9. <button class="btn btn-primary"
  @onclick = "IncrementCount">Click me</button>
10.
11. @code {
12.     private int currentCount = 0;
13.
14.     private void IncrementCount ()
15.     {
16.         currentCount = currentCount + 2;
17.     }
18. }

```

Before saving the file we had the counter page like this:

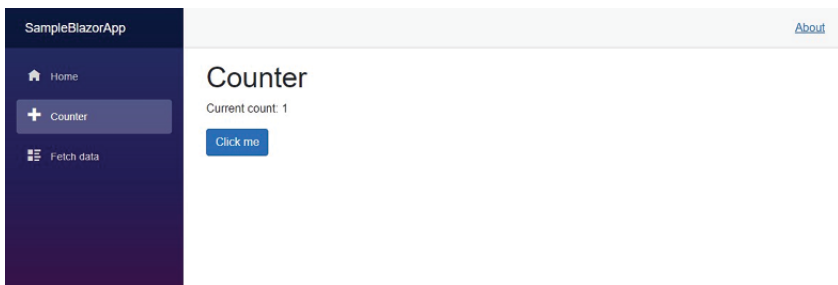


Figure 8.6: The Counter page without changes.

The file was saved, and we now have the page, as shown in [Figure 8.7](#). The page is automatically updated to reflect the code changes without having to rebuild nor re-deploy it.

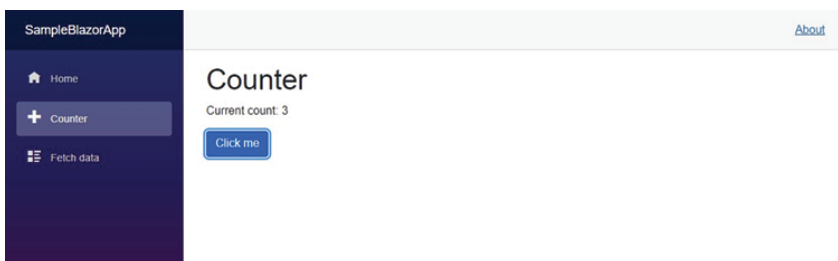


Figure 8.7: The counter page after changes.

Authorization and authentication

In this example we are using the Identity engine managed by a SQL Server instance, accounts and roles are saved in the database. Below are the required steps to have authorization and authentication implemented:

1. Required Nuget Packages:

- Microsoft.AspNetCore.Identity.EntityFrameworkCore
- Microsoft.AspNetCore.Identity.UI
- Microsoft.EntityFrameworkCore
- Microsoft.EntityFrameworkCore.SqlServer
- Microsoft.EntityFrameworkCore.Tools

7. The **Program.cs** had to be updated as follows:

1. `using Microsoft.AspNetCore.Components;`
2. `using Microsoft.AspNetCore.Components.Web;`
3. `using SampleBlazorApp.Data;`
4. `using Microsoft.AspNetCore.Identity;`
5. `using Microsoft.EntityFrameworkCore;`
- 6.
7. `var builder =
 WebApplication.CreateBuilder(args);`
- 8.
9. `builder.Services.AddDbContext<SampleBlazorAppCont`
10. `options => options.UseSqlServer("Data
 Source=DESKTOP-H20012E;Initial
 Catalog=SampleThiagoIdentityDb;Integrated
 Security=True;Connect
 Timeout=30;Encrypt=False;Trust Server
 Certificate=False;Application
 Intent=ReadWrite;Multi Subnet
 Failover=False");`
11. `builder.Services.AddDefaultIdentity<IdentityUser>
 => options.SignIn.RequireConfirmedAccount =
 true).AddEntityFrameworkStores<SampleBlazorAppCor`
- 12.
13. *// Add services to the container.*
14. `builder.Services.AddRazorPages();`

```

15. builder.Services.AddServerSideBlazor();
16. builder.Services.AddSingleton<WeatherForecastService>();
17. var app = builder.Build();
18.
19. // Configure the HTTP request pipeline.
20. if (!app.Environment.IsDevelopment())
21. {
22.     app.UseExceptionHandler("/Error");
23.     // The default HSTS value is 30 days. You may want to
    // change this for production scenarios, see https://aka.ms/
    // aspnetcore-hsts.
24.     app.UseHsts();
25. }
26.
27. app.UseHttpsRedirection();
28.
29. app.UseStaticFiles();
30.
31. app.UseRouting();
32. app.UseAuthentication();
33. app.UseAuthorization();
34.
35. app.MapBlazorHub();
36. app.MapFallbackToPage("/_Host");
37.
38. app.Run();

```

8. The **App** component also had to be updated, as follows:

```

1. <CascadingAuthenticationState>
2. <Router
    AppAssembly="@typeof(App).Assembly">
3.     <Found Context="routeData">
4.         <AuthorizeRouteView
            RouteData="@routeData"
            DefaultLayout="@typeof(MainLayout)" />
5.         <FocusOnNavigate RouteData="@routeData"
            Selector="h1" />

```



```

6.     </Found>
7.     <NotFound>
8.         <PageTitle>Not found</PageTitle>
9.         <LayoutView
Layout="@typeof(MainLayout)">
10.             <p role="alert">Sorry, there's
nothing at this address.</p>
11.         </LayoutView>
12.     </NotFound>
13. </Router>
14. </CascadingAuthenticationState>

```

9. Add a new scaffolded item of type **Identity**. You can find it in the **Identity** sub-menu on the left, as shown in the following figure:

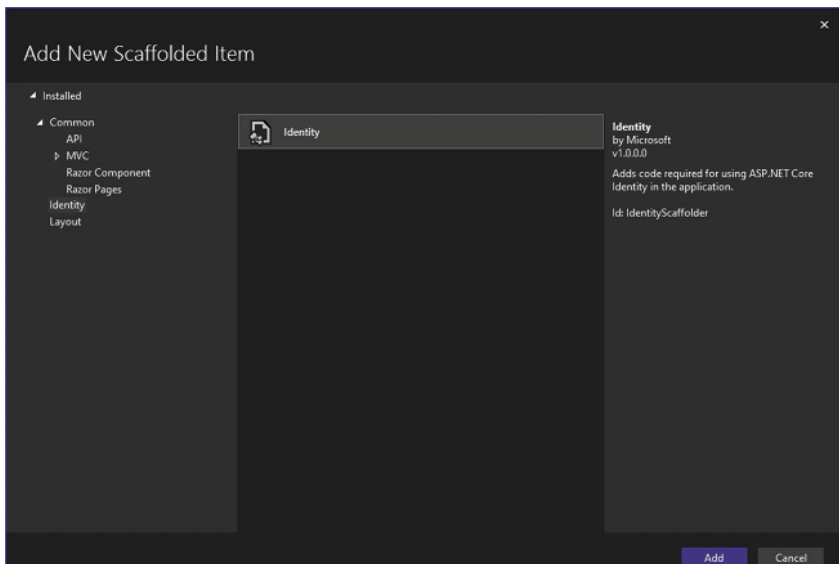


Figure 8.8: Adding an Identity Scaffolded item.

1. Check the **Override all files** option and set your DbContext class:

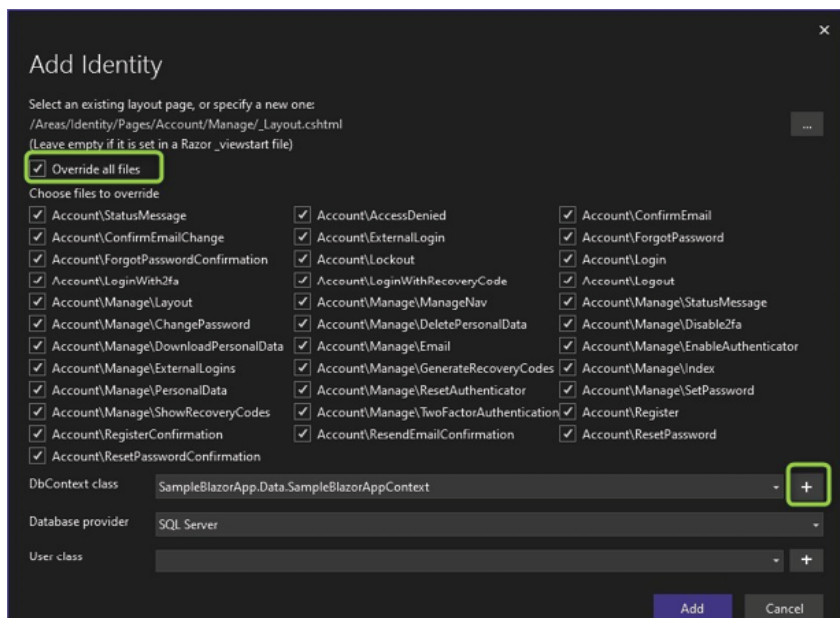


Figure 8.9: Setting up Identity scaffolded items.

We have the following classes created as a result:

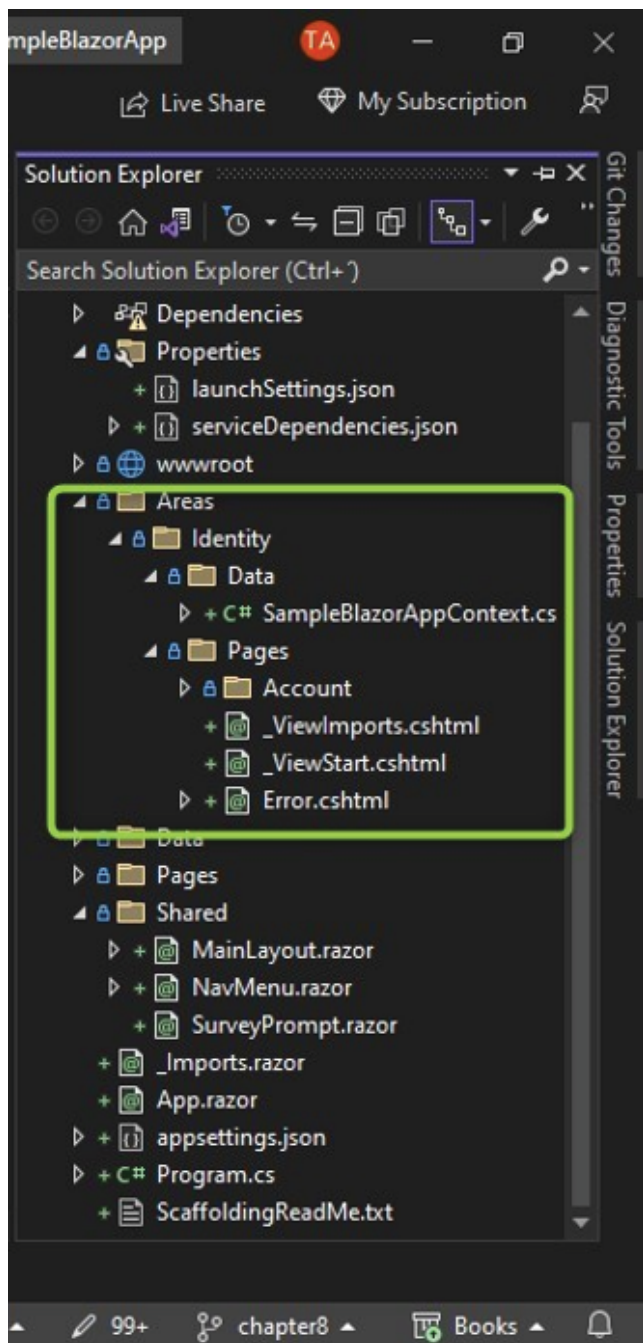


Figure 8.10: The items created after the Identity's scaffolding process.

1. After running the following commands in the Package Manager Console window, we have our Identity set up in the database:
 - a. Add migration
 - b. Update-database

The tables created in the database are responsible for managing user accounts, authentication and authorization:

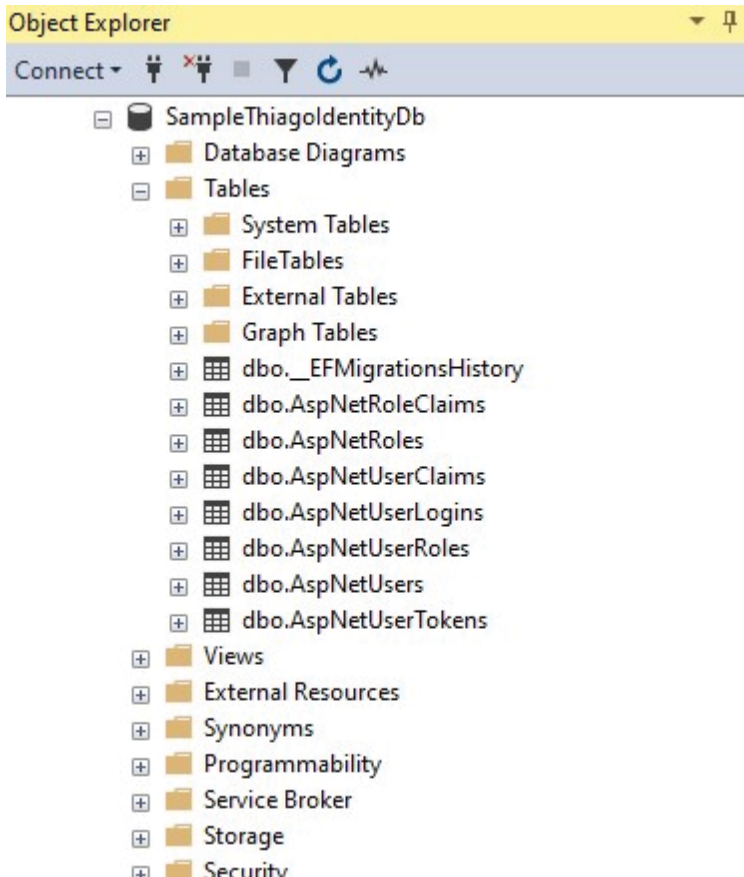


Figure 8.11: The tables created in the database.

1. Change the **FetchData** component, add the **Authorize** attribute, and try to access it.

The code in the **FetchData** component:

```
1. @page "/fetchdata"
```

```

2. @attribute [Authorize]
3. @using SampleBlazorApp.Data
4. @inject WeatherForecastService
   ForecastService
5.
6. <PageTitle>Weather forecast</PageTitle>
7.
8. <h1>Weather forecast</h1>
9.
10. <p>This component demonstrates fetching
    data from a service.</p>
11.
12. @if (forecasts == null)
13. {
14.     <p><em>Loading...</em></p>
15. }
16. else
17. {
18.     <table class="table">
19.         <thead>
20.             <tr>
21.                 <th>Date</th>
22.                 <th>Temp. (C)</th>
23.                 <th>Temp. (F)</th>
24.                 <th>Summary</th>
25.             </tr>
26.         </thead>
27.         <tbody>
28.             @foreach (var forecast in forecasts)
29.             {
30.                 <tr>
31.                     <td>@forecast.Date.ToShortDateString()</
                        td>
32.                     <td>@forecast.TemperatureC</td>
33.                     <td>@forecast.TemperatureF</td>

```

```

34.         <td>@forecast.Summary</td>
35.     </tr>
36. }
37. </tbody>
38. </table>
39. }
40.
41. @code {
42.     private WeatherForecast[]? forecasts;
43.
44.     protected override async Task OnInitializedAsync()
45.     {
46.         forecasts = await
            ForecastService.GetForecastAsync(DateOnly.FromDateTime(DateTime.Now))
47.     }
48. }

```

The output is a Not Authorized message when trying to access the **FetchData** page:

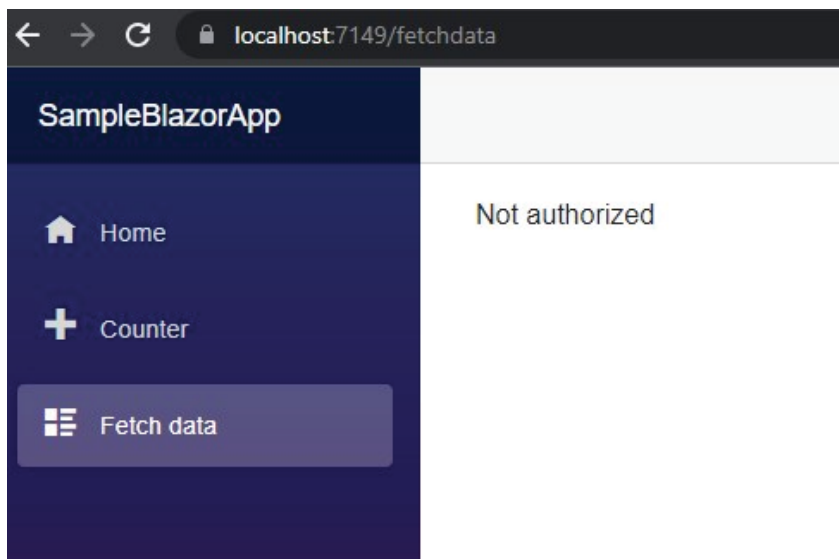
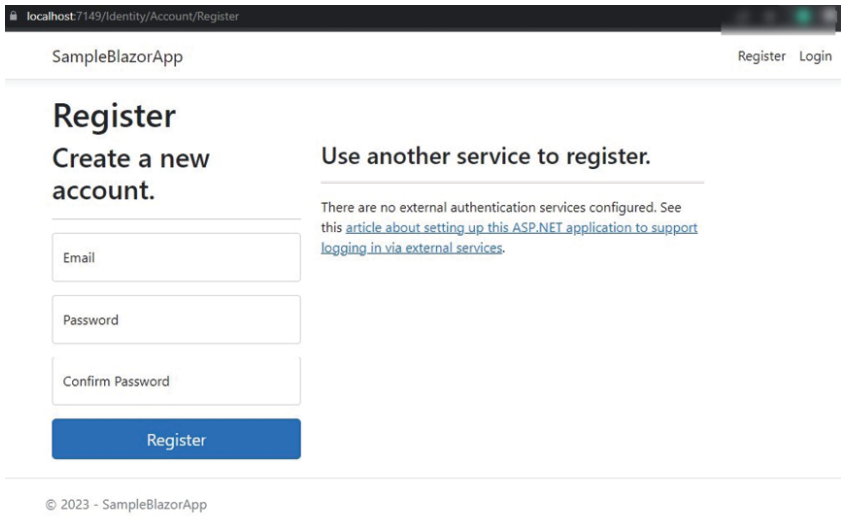


Figure 8.12: The Fetch Data page after using the *Authorize* attribute.

Registering a new account, go to **/Identity/Account/Register**

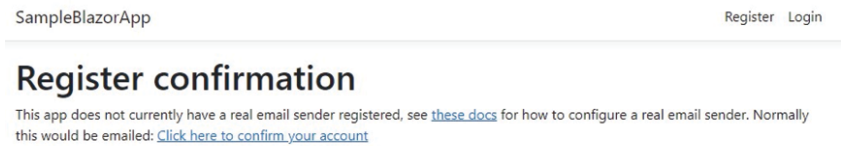
URL:



The screenshot shows the 'Register' page of 'SampleBlazorApp'. The browser address bar shows 'localhost:7149/Identity/Account/Register'. The page has a header with 'SampleBlazorApp' and links for 'Register' and 'Login'. The main content area is titled 'Register' and 'Create a new account.' It contains three input fields: 'Email', 'Password', and 'Confirm Password', followed by a blue 'Register' button. To the right, there is a section 'Use another service to register.' with a message stating that no external authentication services are configured and a link to an article about setting up ASP.NET for external services. The footer shows '© 2023 - SampleBlazorApp'.

Figure 8.13: The page to register a new account.

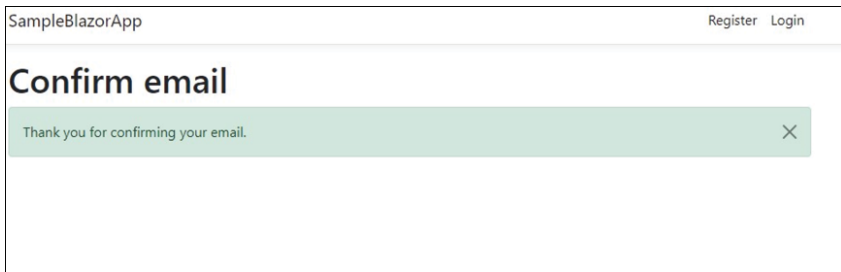
Click on the following link to confirm your account registration:



The screenshot shows the 'Register confirmation' page of 'SampleBlazorApp'. The browser address bar shows 'localhost:7149/Identity/Account/ConfirmEmail'. The page has a header with 'SampleBlazorApp' and links for 'Register' and 'Login'. The main content area is titled 'Register confirmation' and contains a message stating that the app does not currently have a real email sender registered, with a link to 'these docs' for configuration. It also includes a link to 'Click here to confirm your account'.

Figure 8.14: Request confirmation for the account created.

We have created our account successfully:



The screenshot shows the 'Confirm email' page of 'SampleBlazorApp'. The browser address bar shows 'localhost:7149/Identity/Account/ConfirmEmail'. The page has a header with 'SampleBlazorApp' and links for 'Register' and 'Login'. The main content area is titled 'Confirm email' and contains a green success message: 'Thank you for confirming your email.' with a close button (X).

Figure 8.15: Account created and ready to use.

Now, we can access the **FetchData** page again.

Updating the **Counter** component to use the **AuthorizeView**:

```
1. @page "/counter"
2.
3. <PageTitle>Counter</PageTitle>
4.
5. <h1>Counter</h1>
6.
7. <p>
8.     <input @bind="IncrementCount" />
9. </p>
10. <AuthorizeView>
11.     <Authorized>
12.         <p>
13.             @context.User.Identity.Name
14.             <br />
15.             <code>inputValue</code>:
16.             @currentCount
17.         </p>
18.     </Authorized>
19.     <NotAuthorized>
20.         <p>
21.             Not Authorized
22.         </p>
23.     </NotAuthorized>
24. </AuthorizeView>
25. @code {
26.     private int currentCount = 0;
27.
28.     private int IncrementCount
29.     {
30.         get => currentCount;
31.         set => currentCount = currentCount + 1;
32.     }
33. }
```


32. }

In the Authorized response, we can see the `inputValue`:

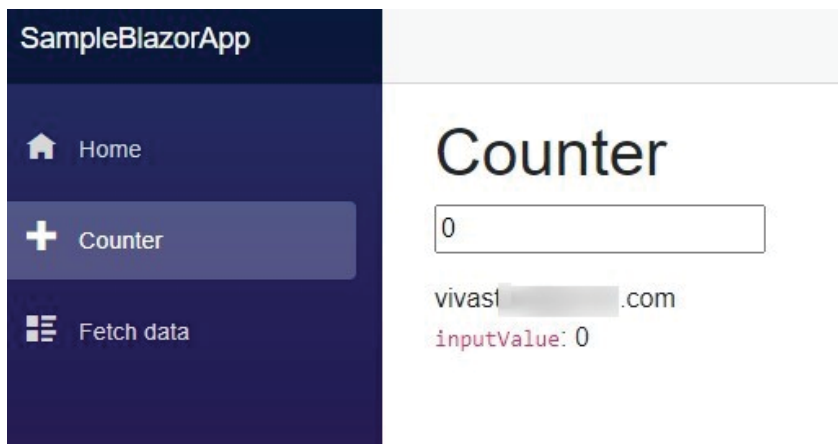


Figure 8.16: The counter page from an Authorized user.

The result is the Not Authorized message for the `inputValue`:

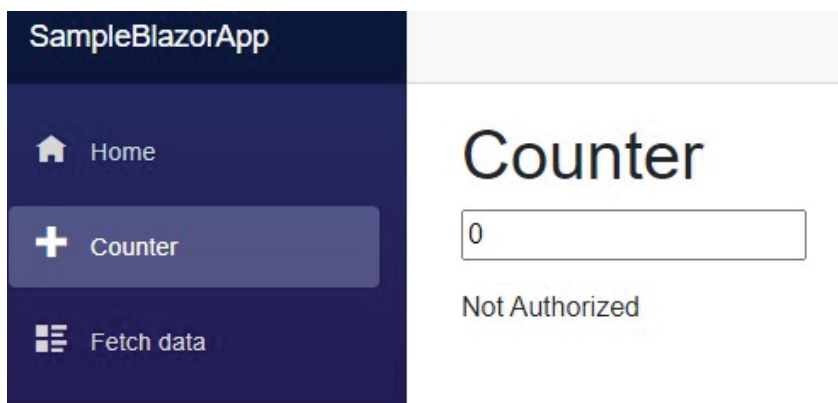


Figure 8.17: The Counter page from a not authenticated user.

Data binding

In the first Data binding example we are applying the two-way data binding with the `@bind:get/@bind:set` modifiers. For this, we are updating the `Counter` component.

This is the `Counter` component after using the `@bind:get/`

@bind:set modifiers:

```
1. @page "/counter"
2.
3. <PageTitle>Counter</PageTitle>
4.
5. <h1>Counter</h1>
6.
7. <p>
8.     <input @bind:event="onchange"
9.         @bind:get="currentCount"
10.         @bind:set="IncrementCount" />
11. </p>
12.
13. <p>
14.     <code>inputValue</code>: @currentCount
15. </p>
16.
17. @code {
18.     private int currentCount = 0;
19.
20.     private void IncrementCount(int value)
21.     {
22.         currentCount = value+1;
23.     }
24. }
```

Setting a number, setting 10 as follows:

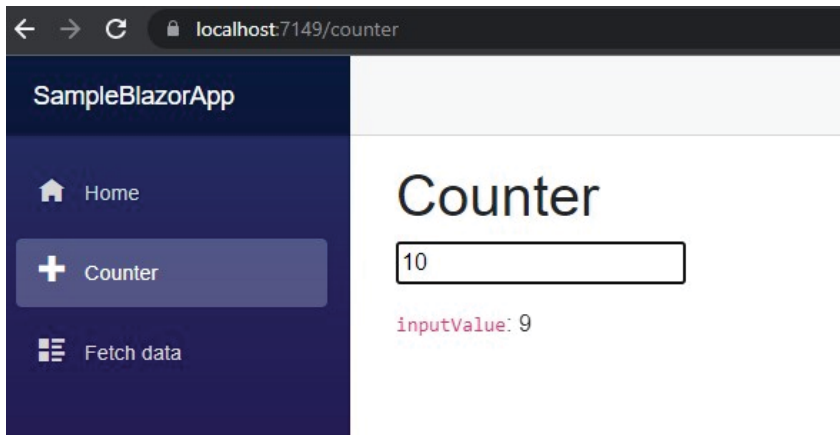


Figure 8.18: The counter page before the event occurs.

The result is 11, as expected:

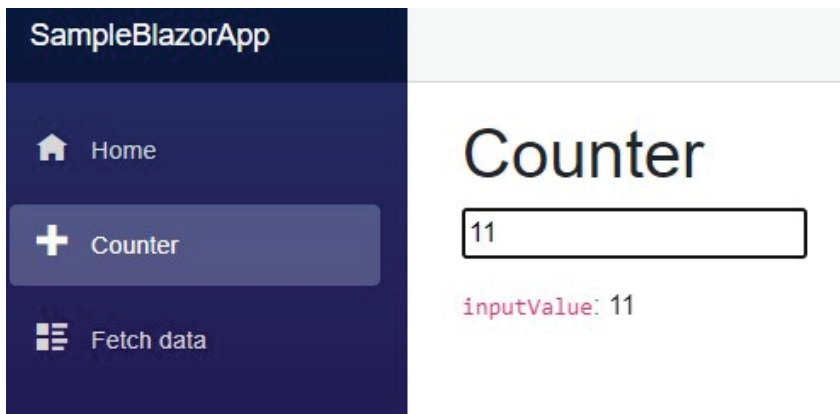


Figure 8.19: The counter page after the event occurs.

Now we are data binding using the C# get and set accessors. Again, we are using the Counter component for this example.

This is the **Counter** component after using the C# get and set accessor. Attention that we are not using the number in the input. We are always incrementing the previous number starting from 0. You can see the whole code block below:

1. `@page "/counter"`
- 2.
3. `<PageTitle>Counter</PageTitle>`

```

4.
5. <h1>Counter</h1>
6.
7. <p>
8.   <input @bind="IncrementCount" />
9. </p>
10.
11. <p>
12.   <code>inputValue</code>: @currentCount
13. </p>
14.
15. @code {
16.   private int currentCount = 0;
17.
18.   private int IncrementCount
19.   {
20.     get => currentCount;
21.     set => currentCount = currentCount + 1;
22.   }
23. }

```

After running the code, we have this:

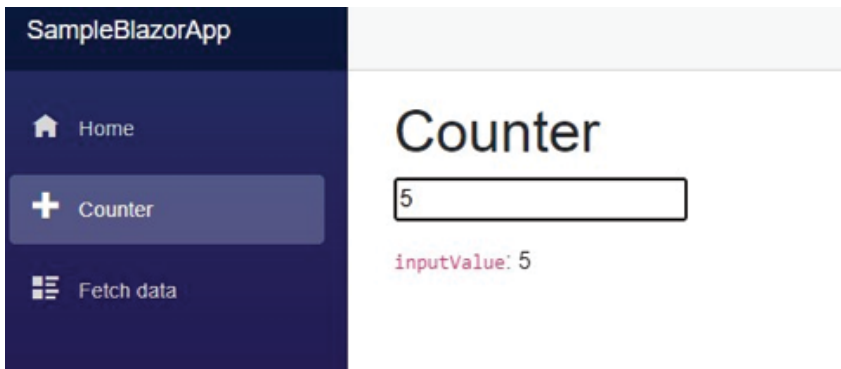


Figure 8.20: The counter page before the event occurs. The values come from the C# objects.

After changing the input value, we have an increment in the previous number. We are not taking into consideration the input

value; we are incrementing the previous `inputValue`:

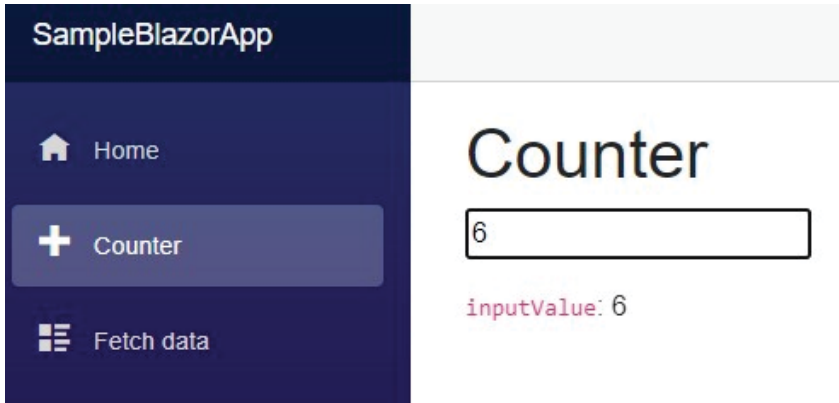


Figure 8.21: The counter page after the event. The values come from the C# objects.

Conclusion

In this chapter, we covered building dynamic web applications with Blazor and ASP.NET. We covered a range of important topics that are fundamental to creating interactive and responsive web apps.

We explored the convenience and productivity of hot reload, allowing developers to make real-time code changes and instantly see the impact. This feature accelerates development cycles and reduces downtime, resulting in a more efficient and enjoyable development experience.

We emphasized the significance of implementing robust security measures in web applications. From authentication and authorization to safeguarding sensitive data, we highlighted the importance of adopting best practices to ensure our applications remain secure and protected against potential threats.

We discovered the benefits of data binding, which promotes type safety and enhances code maintainability. By leveraging this feature, we can bind data between components with confidence, benefiting from compile-time validation and IntelliSense support.

We explored the differences between Blazor and Razor, two powerful technologies for web app development. By understanding

their unique strengths and use cases, we gained insights into choosing the most suitable approach for our specific project requirements.

By mastering these concepts, we are now equipped with the knowledge and skills necessary to develop dynamic web applications with Blazor and ASP.NET. The combination of Blazor's client-side capabilities and ASP.NET's robust web development framework empowers us to create highly interactive and feature-rich applications.

As you continue your journey in web app development, it is essential to explore further and stay updated with the evolving features and advancements in Blazor and ASP.NET. These technologies offer a wealth of possibilities for building cutting-edge web applications that provide engaging user experiences.

We hope this chapter has provided you with valuable insights and practical knowledge to embark on your own dynamic web app projects. Remember to leverage the concepts and techniques covered here to create exceptional web applications that meet the needs of your users.

Thank you for joining us on this exciting exploration of building dynamic web apps with Blazor and ASP.NET. Keep pushing the boundaries of web development and continue to innovate as you embark on your next projects.

In the upcoming chapter, we immerse ourselves in the world of *Real-Time Communication with SignalR and ASP.NET*. This topic marks a pivotal exploration of how SignalR, a robust library for real-time web functionality, seamlessly integrates with ASP.NET to enable dynamic, two-way communication between clients and servers. As we navigate through the intricacies of real-time communication, readers will uncover the power of SignalR in creating interactive and responsive web applications. Through practical demonstrations and insights, this chapter aims to equip readers with the knowledge to implement real-time features, transforming the traditional web experience into a dynamic and engaging platform.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech

happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 9

Real-time Communication with SignalR and ASP.NET

Introduction

In today's fast-paced digital world, users expect applications to deliver real-time updates and interactions. Whether it is a live chat, collaborative editing, or dynamic data visualization, the ability to provide real-time communication is crucial for building engaging and interactive applications.

In this chapter, we will explore SignalR, a powerful framework provided by ASP.NET, which simplifies the process of adding real-time capabilities to your applications. With SignalR, you can easily establish bidirectional communication channels between the server and the client, enabling seamless and instantaneous data exchange.

We will begin by diving into the configuration aspects of SignalR. You will learn how to set up and configure SignalR in your ASP.NET application, ensuring you have a solid foundation to build. We will explore various options and settings that allow you to customize the

behavior of SignalR to suit your specific needs.

Next, we will delve into the crucial aspects of authentication and authorization in real-time communication scenarios. You will discover how to secure your SignalR connections, ensuring only authorized users can access and interact with your real-time features. We will explore different authentication mechanisms and explore how to implement them effectively in your application.

Once we have a solid understanding of the fundamentals, we will explore the Streaming Hub feature of SignalR. Streaming Hubs allow you to build real-time streaming scenarios where clients can receive continuous streams of data from the server. We will examine how to implement streaming hubs, handle large data sets, and optimize performance for efficient streaming.

Structure

This chapter covers the following topics:

- Real-time communication with SignalR and ASP.NET
- Configuration
- Authentication and authorization
- Streaming hub
- Case study

Objectives

By the chapter's end, you will grasp crucial aspects of SignalR and ASP.NET for real-time communication, covering configuration intricacies, authentication, and authorization essentials. Configuration insights include SignalR setup, exploring customization options. Authentication and authorization significance are unraveled, guiding the implementation of secure SignalR connections and control over real-time feature access. Streaming Hubs are demystified, covering client-to-server and server-to-client streaming. In a culmination, a comprehensive case study showcases real-world implementation, offering a step-by-step guide through a practical application. Readers will gain a profound understanding of SignalR's capabilities, from initial setup to secure communication and practical application integration.

Real-time communication with SignalR and ASP.NET

SignalR is an open-source library specifically designed for developing real-time web applications. Its primary purpose is to facilitate instant content delivery from the server to clients, as well as enabling clients to send content to the server in real-time.

By utilizing SignalR, developers can build interactive and responsive web applications that provide seamless and instantaneous updates to connected clients. This two-way communication capability allows for efficient data exchange, enabling real-time collaboration, live updates, and interactive user experiences.

Where to use SignalR

SignalR is well-suited for various types of applications that require real-time communication and frequent updates. Here are some examples of where SignalR can be beneficial:

- **Apps with high-frequency updates:** SignalR is ideal for applications that involve real-time data updates, such as online games, social networks, mapping, and GPS applications. It allows for instant delivery of updates to all connected clients, providing a smooth and interactive user experience.
- **Collaborative apps:** SignalR enables real-time collaboration in applications like whiteboard or team meeting apps. Multiple users can work together simultaneously, seeing each other's changes in real-time and facilitating seamless collaboration.
- **Apps with notifications:** SignalR can be used to send instant notifications to connected clients, making it suitable for applications that require real-time alerts and updates. This includes email applications, chat systems, and any other scenario where immediate notifications are crucial.
- **Any app that requires instant communication:** SignalR can be applied to a wide range of applications where instant communication between the server and clients is essential. This includes real-time monitoring systems, financial trading platforms, live sports updates, and more.

In summary, SignalR is versatile and can be used in various scenarios where real-time communication, frequent updates, and

instant content delivery are required. Its capabilities empower developers to create interactive and responsive applications across different domains.

Message transports

SignalR supports multiple transportation methods to establish a real-time connection between the client and the server. The following transport methods are supported by SignalR:

- **WebSockets:** It provides a full-duplex communication channel between the client and the server, allowing for efficient real-time data transfer with low latency.
- **Server-Sent Events (SSE):** It is a unidirectional communication method where the server sends continuous updates to the client over a single HTTP connection.
- **Long polling:** Long polling is a technique where the client sends a request to the server and keeps the connection open until the server has new data to send. Once the server responds with the data or a timeout occurs, the client immediately sends a new request.

Hubs

Hubs are the core component of SignalR that manages the communication between the client and the server. When the backend server receives a message for a hub, it automatically invokes the corresponding client-side code based on the method name provided in the message.

SignalR supports two built-in Hub Protocols for transporting messages:

- **Text protocol based on JSON:** Messages are serialized in JSON format before being sent to the clients.
- **Binary protocol based on MessagePack:** Messages are serialized using the efficient MessagePack format, which reduces the payload size and improves performance.

SignalR methods

SignalR methods are used to send messages between the server and connected clients. Here are some commonly used methods:

- **Clients.All.SendAsync**: Invokes the specified method on the hub and sends the objects to all connected clients.
- **Clients.Caller.SendAsync**: Invokes the specified method on the hub and sends the objects back to the calling client itself.
- **Clients.Client({connectionId}).SendAsync**: Invokes the specified method on the hub and sends the objects to the specified client based on their connection ID.
- **Groups.AddToGroupAsync**: Adds a client to the specified group, allowing targeted message delivery to clients belonging to that group.
- **Clients.Group({group}).SendAsync**: Invokes the specified method on the hub and sends the objects to all clients in the specified group.

These methods enable efficient communication and message delivery between the server and clients in various scenarios.

Overall, SignalR offers flexible message transports, powerful hubs for managing communication, and convenient methods for sending messages to specific clients or groups of clients.

Configuration

Configuration plays a vital role in harnessing the full potential of SignalR for real-time communication. By understanding and utilizing the various configuration options, developers can fine-tune the behavior of SignalR to align with their application's specific needs. From establishing connection settings and managing hub configuration to optimizing performance and scalability, mastering SignalR's configuration allows for seamless integration and efficient utilization of real-time capabilities in cloud, web, and desktop applications. Whether it is adjusting message size limits, enabling transport protocols, or configuring connection timeouts, a well-configured SignalR environment ensures reliable and high-performance real-time communication between server and client, creating engaging and interactive user experiences.

ASP.NET Core SignalR offers support for two protocols when encoding messages: JSON and MessagePack. Each protocol comes with its own set of serialization configuration options, providing flexibility in adapting to specific requirements and optimizing

message encoding for efficient communication.

JSON encoding

Within the SignalR Protocol's JSON Encoding, every message is represented as a standalone JSON object, serving as the sole content of the underlying transport message. It is important to note that all property names in the JSON object are case-sensitive. The responsibility of encoding and decoding the text lies with the underlying protocol implementation, requiring the JSON string to be encoded in the format expected by the specific transport used. For instance, when employing the ASP.NET Sockets transports, UTF-8 encoding is consistently utilized for text representation.

To configure JSON serialization on the server side in SignalR, you can utilize the **AddJSONProtocol** extension method. This method is added after the **AddSignalR** method within the **ConfigureServices** method in the Startup class. When using **AddJSONProtocol**, you can pass a delegate that receives an options object, and within that object, you can access the **PayloadSerializerOptions** property. This property allows you to configure the serialization of arguments and return values using a **System.Text.JSON JSONSerializerOptions** object.

MessagePack encoding

MessagePack is a highly efficient and compact binary serialization format, ideal for scenarios where performance and bandwidth optimization are crucial. Compared to JSON, MessagePack produces smaller message sizes, making it beneficial in reducing network traffic. However, due to its binary nature, the messages appear unreadable when inspecting network traces and logs unless they are parsed through a MessagePack parser. SignalR acknowledges the significance of MessagePack and incorporates built-in support for this format, offering dedicated APIs for both the client and server to seamlessly utilize MessagePack serialization.

To activate the MessagePack Hub Protocol on the server-side, simply install

Microsoft.AspNetCore.SignalR.Protocols.MessagePack package within your application. In the **Startup.ConfigureServices** method, include the

AddMessagePackProtocol method alongside the **AddSignalR** call to enable seamless MessagePack support on the server.

Currently, it is not feasible to configure MessagePack serialization within the JavaScript client.

Server configuration options

This section explores various configuration options that empower developers to achieve optimal performance and customization. From controlling transport protocols to authentication and authorization settings, learn how to optimize your server applications for efficiency and adaptability. Unleash the potential of server configuration options and elevate your applications to new levels of control and functionality as you can see the main options below:

- **ClientTimeoutInterval:**

- The server considers a client disconnected if it has not received any messages within a specific interval. It is recommended to set the timeout interval to double the for more reliable disconnection detection.
- Default value: 30s

- **HandshakeTimeout:**

- Adjust the handshake timeout interval only in cases of severe network latency causing handshake timeout errors.
- Default value: 15s

- **KeepAliveInterval:**

- To ensure that the connection remains open, a ping message is automatically sent if the server has not sent any messages within a specific interval. When adjusting the **KeepAliveInterval**, it is advisable to also update the **ServerTimeout** or **serverTimeoutInMilliseconds** setting on the client. The recommended value for the **ServerTimeout** or **serverTimeoutInMilliseconds** is double the **KeepAliveInterval** to maintain a reliable and uninterrupted connection.
- Default value: 15s

- **SupportedProtocols:**

- The hub supports multiple protocols, and by default, all registered protocols on the server are allowed. However, it is possible to disable specific protocols for individual hubs by removing them from the list of supported protocols. This provides flexibility in customizing protocol support based on the requirements of each specific hub.
- **EnableDetailedErrors:**
 - By default, detailed exception messages are not returned to clients in Hub methods to safeguard sensitive information.
- **StreamBufferCapacity:**
 - The maximum number of items that can be buffered for client upload streams determines the limit for buffering. When this limit is reached, the processing of invocations is temporarily blocked until the server processes the stream items. This helps maintain a controlled flow of data and prevents overwhelming the server with excessive buffering.
 - Default value: 10
- **MaximumReceiveMessageSize:**
 - The maximum size of a single incoming hub message refers to the limit imposed on the message size. Increasing this value may raise the risk of a **Denial-of-Service (DoS)** attack, as larger messages require more processing resources. It is crucial to strike a balance between accommodating legitimate message sizes and mitigating the potential risks associated with larger payloads.
 - Default value: 32kb
- **MaximumParallelInvocationsPerClient:**
 - The maximum number of concurrent hub method calls from a client before queuing.
 - Default value:1
- **DisableImplicitFromServicesParameters:**
 - If the required dependencies for a hub method are registered in the DI container, they will be automatically resolved and provided as arguments to the method during invocation. This simplifies the process of accessing and utilizing dependencies within hub methods, promoting modularity and maintainability in your application.

- Default value: false

Advanced HTTP configuration

To configure advanced settings concerning transports and memory buffer management, you can utilize the **HttpConnectionDispatcherOptions**. These options can be customized by passing a delegate to the **MapHub** method in the **Program.cs** file. By accessing the **HttpConnectionDispatcherOptions**, you gain control over various aspects of your application's behavior, allowing for fine-tuning and optimization based on specific requirements as you can see the main optimization settings below:

- **ApplicationMaxBufferSize:**

- The server's buffering of client-received bytes before applying backpressure is determined by the maximum number of bytes set. If you increase this value, the server can receive larger messages at a faster pace without applying backpressure, but it may also lead to higher memory consumption. It is important to strike a balance between receiving larger messages quickly and managing memory efficiently.
- Default value: 64kb

- **TransportMaxBufferSize:**

- The server's buffering of app-sent bytes, before observing backpressure, is determined by the maximum number of bytes set. If you increase this value, the server can buffer larger messages more rapidly without waiting for backpressure, but it may also result in higher memory consumption. It is essential to find a balance between buffering larger messages efficiently and managing memory consumption effectively.
- Default value: 64kb

- **AuthorizationData:**

- The list of **IAuthorizeData** objects determines client authorization to connect to the hub by enforcing specified authorization requirements.

- **Transports:**

- By default, all transports are enabled, meaning there are no

restrictions on which transports a client can use to establish a connection.

- **LongPolling:**

- Specific options are available for the Long Polling transport in SignalR.

- **PollTimeout**

- The maximum wait time for the server to send a message to the client before terminating a single poll request can be configured. Reducing this value results in the client issuing poll requests more frequently, enabling faster message delivery.

- Default value: 90s

- **WebSockets:**

- Default value:

- **CloseTimeout:**

- When the server closes, there is a specified time interval within which the client should also close the connection. If the client fails to close within this interval, the connection is terminated by the server.

- Default value: 5s

- **SubProtocolSelector:**

- A delegate can be utilized to set the Sec-WebSocket-Protocol header to a custom value. This delegate receives the requested values from the client as input and is responsible for returning the desired value for the Sec-WebSocket-Protocol header. By using this delegate, you have the flexibility to customize and control the value of the Sec-WebSocket-Protocol header based on your specific requirements.

- Default value: Null

- **MinimumProtocolVersion:**

- Specify the minimum negotiate protocol version to restrict clients to newer versions.
- Default value: 0

- **CloseOnAuthenticationExpiration:**

- Enabling authentication expiration tracking allows for the automatic closure of connections when a token expires. By setting this option, you ensure that connections are terminated promptly when authentication tokens expire, enhancing security and preventing unauthorized access.
- Default value: false

Client configuration options

These options enable developers to optimize performance, enhance security, and tailor the client-side experience to specific requirements. From connection management and transport protocols to handling retries and timeouts, these configuration options provide flexibility and control over the SignalR client's behavior.

By delving into the intricacies of client configuration options, you gain the ability to optimize network communication, streamline error handling, and adapt the client-side behavior to align with your application's needs.

Configure logging

In the .NET Client, logging configuration is achieved through the **ConfigureLogging** method. This method allows you to register logging providers and filters, similar to how they are configured on the server. By utilizing **ConfigureLogging**, you can customize the logging behavior of your client application, enabling comprehensive monitoring, troubleshooting, and analysis. With the flexibility to choose and configure logging providers and filters, you can effectively manage and track client-side activities, ensuring optimal performance and addressing potential issues, as we can see an example of logging configuration below:

```
1. var connection = new HubConnectionBuilder()  
2.     .WithUrl("https://example.com/chathub")  
3.     .ConfigureLogging(logging => {  
4.  
       logging.SetMinimumLevel(LogLevel.Information);  
5.       logging.AddConsole();  
6.     })
```

```
7. .Build();
```

Configure allowed transports

The transports used by SignalR can be configured during the **WithUrl** call (or **withUrl** in JavaScript). By performing a bitwise-OR operation on the values of **HttpTransportType**, you can specify which transports the client is allowed to use. By default, all transports are enabled, providing flexibility and compatibility. However, by customizing the transport configuration, you can restrict the client to only use the specified transports, ensuring optimal performance and targeted network communication based on your application's requirements, as you can see in the following code example:

```
1. var connection = new HubConnectionBuilder()
2.   .WithUrl("https://example.com/chathub",
3.     HttpTransportType.WebSockets |
4.     HttpTransportType.LongPolling)
5.   .Build();
```

Configure Bearer authentication

To include authentication data in SignalR requests, use the **AccessTokenProvider** option (**accessTokenFactory** in JavaScript). In the .NET Client, the access token is passed as a "Bearer Authentication" token using the Authorization header. In the JavaScript client, the access token is used as a Bearer token, except in cases where browser APIs restrict header usage, such as Server-Sent Events and WebSockets requests. In these cases, the access token is provided as a query string parameter named **access_token**, as you can see the authentication configuration in the following code example:

```
1. var connection = new HubConnectionBuilder()
2.   .WithUrl("https://example.com/chathub",
3.     options => {
4.       options.AccessTokenProvider = async
5.         () => {
6.           // Get and return the access token.
7.         };
8.     })
```

Additional options

SignalR provides a comprehensive set of client options that go beyond the basic configuration, enabling you to fine-tune and enhance the behavior of your SignalR clients. These additional client options empower you to optimize performance, customize connectivity, and adapt to specific requirements, as you can see the main options below:

- **ServerTimeout:**

- To establish a robust connection, it is crucial to set an appropriate server activity timeout. This value should be carefully chosen to allow ample time for ping messages sent from the server to be successfully transmitted and received by the client within the specified timeout interval. It is recommended to set the server activity timeout to a value that is at least twice the server's `KeepAliveInterval`. This additional buffer allows sufficient time for the ping messages to reach the client, minimizing the risk of premature disconnection.
- Default value: 30s

- **HandshakeTimeout :**

- Modifying the initial server handshake timeout is considered an advanced setting, suitable only for cases where handshake timeout errors occur due to significant network latency. It is crucial to exercise caution when adjusting this timeout and do so only when necessary.
- Default value:15s

- **KeepAliveInterval:**

- In SignalR, the client keep-alive interval governs the frequency at which the client sends ping messages to the server. These ping messages serve as a crucial mechanism to maintain the health of the connection. Whenever the client sends any message, the keep-alive timer resets, initiating a new interval.
- Default value: 15s

- **AccessTokenProvider:**

- A Bearer authentication token in HTTP requests can be

obtained by invoking a function that returns a string. This string, once provided as the authentication token, enables secure and authorized access to protected resources.

- Default value: null

- **SkipNegotiation:**

- Enabling this option bypasses the negotiation step and directly establishes a connection using the WebSockets transport. It is important to note that this feature is only supported when the WebSockets transport is the sole enabled transport. However, please be aware that this setting cannot be enabled when utilizing the Azure SignalR Service

- Default value: false

- **ClientCertificates:**

- A collection of TLS certificates that can be included to authenticate requests, ensuring secure and trusted communication.

- Default value: empty

- **Cookies:**

- A collection of HTTP cookies that can be included with every HTTP request to provide additional information or maintain stateful communication.

- Default value: empty

- **Credentials:**

- Credentials can be sent with each HTTP request to ensure authentication and authorization for accessing protected resources.

- Default value: empty

- **CloseTimeout:**

- In the context of WebSockets, the client can specify a maximum timeout for the server to acknowledge a close request after the client initiates the closing process. If the server fails to acknowledge the close within this specified time, the client will disconnect from the server. This timeout setting helps ensure a timely termination of the WebSocket connection, preventing prolonged waits and allowing for smooth and efficient communication.

- Default value: 5s
- Headers:
 - A map of additional HTTP headers that can be included with each HTTP request to provide extra information or customization.
 - Default value: empty
- **HttpMessageHandlerFactory:**
 - The **HttpMessageHandler** delegate is used to configure or replace the HTTP message handler for sending requests. It receives the default handler as a parameter and must return a non-null value. You can modify the settings on the default handler and return it, or create a new handler instance. Make sure to copy any necessary settings from the provided handler to the new one for proper functionality.
 - Default value: null
- **Proxy:**
 - The proxy to be used.
 - Default value: null
- **UseDefaultCredentials:**
 - If any default credentials are to be used, enable the use of Windows authentication.
 - Default value: false
- **WebSocketConfiguration:**
 - A delegate is used to configure additional WebSocket options. It receives an instance of **ClientWebSocketOptions** that can be used to customize the options.
 - Default value: null
- **ApplicationMaxBufferSize:**
 - Increasing the value of this option allows the application to buffer a larger number of bytes received from the server before applying backpressure. This can result in faster processing of larger messages, but it may also increase memory consumption.
 - Default value: 1mb

- **TransportMaxBufferSize:**

- Increasing the value of this option allows the transport to buffer a larger number of bytes sent by the user application before observing backpressure. This can result in faster buffering of larger messages, but it may also increase memory consumption.
- Default value: 1mb

Authentication and authorization

SignalR can leverage ASP.NET Core authentication to establish a connection-user association. This allows authentication data to be accessed from the **HubConnectionContext.User** property within a hub. By utilizing authentication, hubs gain the ability to invoke methods on all connections associated with a particular user. It is worth noting that multiple connections can be linked to a single user.

When a hub method requires authorization, SignalR offers a custom resource to authorization handlers. This resource is represented by an instance of **HubInvocationContext**. The **HubInvocationContext** provides access to various information, including the **HubCallerContext**, which contains details about the connection and user associated with the invocation. Additionally, it includes the name of the hub method being invoked and the arguments passed to that method. This context serves as a valuable resource for performing authorization checks and making informed decisions within your SignalR application.

Cookie authentication

In browser-based applications, cookie authentication enables seamless integration of existing user credentials with SignalR connections. When using the browser client, no additional configuration is required. If the user is logged in to the application, the SignalR connection automatically inherits this authentication.

Cookies serve as a browser-specific mechanism for transmitting access tokens, but they can also be used by non-browser clients. In the .NET Client, you can configure the **Cookies** property within the **WithUrl** method to provide a cookie for authentication. However, utilizing cookie authentication from the .NET client necessitates the

application to offer an API endpoint for exchanging authentication data in order to obtain the required cookie. This enables seamless authentication between the client and server components of the application.

Bearer token authentication

The server validates the token and utilizes it to identify the associated user. This validation process occurs solely during the establishment of the connection. Once the connection is established, the server does not automatically revalidate the token to check for token revocation during the connection's lifespan. Therefore, it is crucial to ensure the access token remains valid throughout the duration of the connection to maintain uninterrupted authentication, as we can see in the following connection with the SignalR connection:

```
1. var connection = new HubConnectionBuilder()
2.    .WithUrl("https://example.com/chathub",
3.    options =>
4.    {
5.        options.AccessTokenProvider
6.        = () =>
7.        Task.FromResult(_myAccessToken);
8.    })
9.    .Build();
```

Identity server JWT authentication

The identity server is configured on the server, using the JWT bearer middleware as you can see in the following example:

```
1. builder.Services.AddAuthentication(options
2. =>
3. {
4.    options.DefaultAuthenticateScheme =
5.    JwtBearerDefaults.AuthenticationScheme;
6.    options.DefaultChallengeScheme =
7.    JwtBearerDefaults.AuthenticationScheme;
8. }
```



```

7. }) .AddJwtBearer(options =>
8. {
9.     options.Authority = "Authority URL";
10.
11.     options.Events = new JwtBearerEvents
12.     {
13.         OnMessageReceived = context =>
14.         {
15.             var accessToken =
context.Request.Query["access_token"];
16.
17.             var path =
context.HttpContext.Request.Path;
18.             if (!
string.IsNullOrEmpty(accessToken) &&
19. (path.StartsWithSegments("/hubs/chat")))
20.             {
21.                 context.Token =
accessToken;
22.             }
23.             return Task.CompletedTask;
24.         }
25.     };
26. });

```

Windows authentication

If Windows authentication is configured in the app, SignalR can utilize that identity to secure hubs. However, to send messages to individual users, a custom user ID provider needs to be added. It is important to note that the Windows authentication system does not provide the name identifier claim, which SignalR relies on to determine the username.

Please be aware that while Windows authentication is supported in Microsoft Edge, it may not be supported in all browsers. For example, attempting to use Windows authentication and

WebSockets in Chrome and Safari will result in failure. In such cases, the client will attempt to fallback to other transports that might work, as you can see in the following example:

```
1. var connection = new HubConnectionBuilder()
2.     .WithUrl("https://example.com/chathub",
3.         options =>
4.         {
5.             options.UseDefaultCredentials =
6.                 true;
7.         })
8.     .Build();
```

Claims

To derive SignalR user IDs from user claims in an app that authenticates users, you can implement the **IUserIdProvider** interface and register the implementation. By implementing this interface, you can specify how SignalR creates user IDs based on the user's claims, as you can see in the following example where we get the email value from the user claims:

```
1. public class SampleEmailProvider :
2.     IUserIdProvider
3. {
4.     public virtual string
5.         GetUserId(HubConnectionContext
6.             connection)
7.     {
8.         return
9.             connection.User?.FindFirst(ClaimTypes.Email)?.
10.                 Value!;
```

Policies for hubs and hubs methods authorization

To enforce authentication for methods in a hub, you can apply the **[Authorize]** attribute to the hub itself. By default, all methods in the hub will require authentication to be accessed. This attribute ensures that only authenticated users can invoke hub methods.

Additionally, you can customize the `[Authorize]` attribute by providing constructor arguments and properties to restrict access to users who match specific authorization policies. For example, you can specify a custom authorization policy named `SampleAuthorizationPolicy`, and only users who meet that policy's requirements will be allowed to access the hub.

Furthermore, if you want to apply authorization on a per-method basis, you can apply the `[Authorize]` attribute directly to individual hub methods. In this case, if the current user does not satisfy the policy applied to the method, an error will be returned to the caller, indicating the lack of authorization.

By leveraging the `[Authorize]` attribute, you can enforce authentication and authorization rules to control access to hub methods and ensure that only authorized users can invoke them.

```
1. [Authorize("SampleAuthorizationPolicy")]
2. public class SampleHub : Hub
3. {
4.     public override async Task
       OnConnectedAsync()
5.     {
6.         await
           Clients.All.SendAsync("ReceiveSystemMessage",
7.                                 $"{Context.UserIdIdentifier}
              joined.»");
8.         await base.OnConnectedAsync();
9.     }
10.    [Authorize("Administrators")]
11.    public void BanUser(string userName)
12.    {
13.        // ... ban a user from the chat room (something
           only Administrators can do) ...
14.    }
15. }
```

Authorization handlers

By default, SignalR provides a robust built-in authorization framework that allows you to authenticate and authorize users.

However, there may be scenarios where you need to implement custom logic to enforce additional access restrictions or business rules. This is where custom authorization policies come in.

Custom authorization policies empower you to define your own rules and criteria for granting or denying access to specific features or functionality within your SignalR application. With custom policies, you have the flexibility to implement fine-grained control over who can perform certain actions, such as sending messages, joining specific groups, or accessing sensitive data.

Streaming hub

In ASP.NET Core SignalR, you can utilize streaming to enable data transmission between the client and the server in fragments or chunks. Streaming is particularly beneficial in scenarios where data arrives gradually over time, allowing each fragment to be sent to the client or server as soon as it becomes available.

With streaming, you do not need to wait for the entire data set to be ready before sending it. Instead, you can start sending fragments of data as they become available, providing a more responsive and efficient communication mechanism.

This capability enables real-time data streaming and processing, making it suitable for various use cases such as live updates, media streaming, and handling large data sets.

By leveraging the streaming feature in ASP.NET Core SignalR, you can achieve more efficient data transmission, improved responsiveness, and enhanced real-time capabilities in your applications.

Server-to-client streaming hub

ASP.NET Core SignalR allows you to perform streaming between the server and the client. You have the option to return an **`IAsyncEnumerable<T>`** or a **`ChannelReader<T>`** from your streaming hub methods.

To return an **`IAsyncEnumerable<T>`**, you can make your hub method an async iterator method. This approach simplifies the implementation and ensures that the **`ChannelReader`** is returned appropriately. You can also include a **`CancellationToken`**

parameter in the async iterator method to handle client unsubscriptions.

When using server-to-client streaming hub methods, it is important to handle client disconnections. By accepting a **CancellationToken** parameter, you can detect when the client unsubscribes from the stream. You can use this token to gracefully stop the server operation and release any associated resources.

Using these techniques, you can effectively stream data between the client and the server, ensuring proper handling of client subscriptions, unsubscriptions, and disconnections.

Client-to-server streaming hub

When a hub method accepts one or more objects of type **ChannelReader<T>** or **IAsyncEnumerable<T>**, it automatically becomes a client-to-server streaming hub method. This allows the client to send streaming data to the server.

In the following sample, you can see how to read the streaming data sent from the client. The client writes data to the **ChannelWriter<T>**, and the hub method on the server reads this data from the corresponding **ChannelReader<T>**.

By leveraging this functionality, you can establish a streaming connection between the client and the server, enabling efficient and continuous data transfer.

Case study

Welcome to the case study, where we will apply the concepts of SignalR configuration, authentication and authorization, and custom policy in the development of a real-time chat application. This case study will provide you with a practical demonstration of how these elements come together to create a secure and efficient real-time communication platform.

Throughout this case study, we will focus on building a robust chat application that enables users to exchange messages in real-time. We will start by configuring SignalR in our ASP.NET application, ensuring that we have a solid foundation for real-time communication. This includes setting up the necessary hubs, establishing connections, and defining the desired behavior.

Next, we will address the crucial aspect of authentication and authorization. We will implement a secure login system and explore Entity Framework Identity authentication mechanism available with SignalR. We will also delve into custom policy creation to control access to the chat application, ensuring that only authenticated users can participate.

To enhance the user experience and maintain the integrity of the chat, we will implement a sample custom policy. We will explore the implementation of these policies using SignalR's built-in authorization framework, empowering you to tailor the chat application to your specific needs.

So, let us embark on this case study journey, where we will bring together the power of SignalR, authentication, authorization, and custom policy to create a secure and feature-rich real-time chat application. Get ready to dive into the exciting world of real-time communication with SignalR!

First, we create a project of type Web Application and add a client-side library `@microsoft/signalr@latest`

To do this, you should right-click on the project | Add | Client-side library. Search for the SignalR library as shown in the figure below:

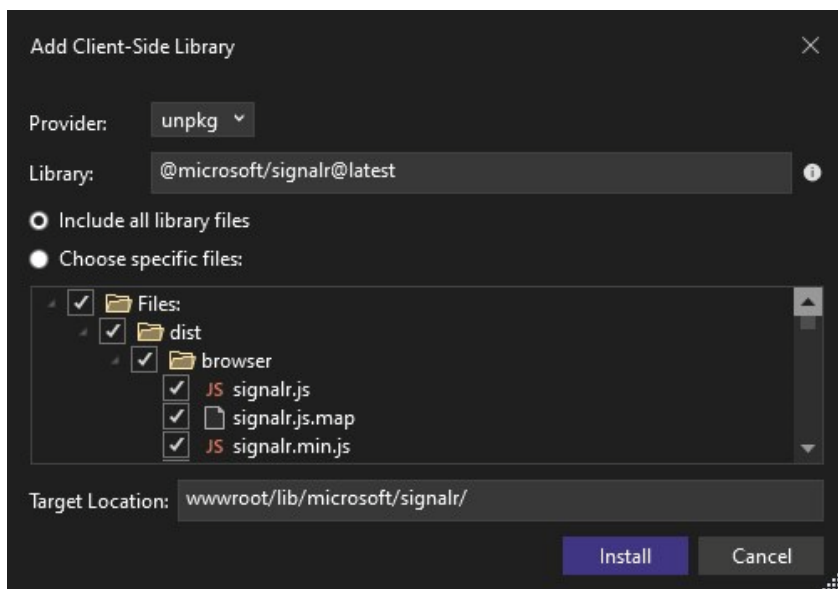


Figure 9.1: Adding SignalR client-side library.

After successfully adding the client-side library then, you must have your project like the following figure:

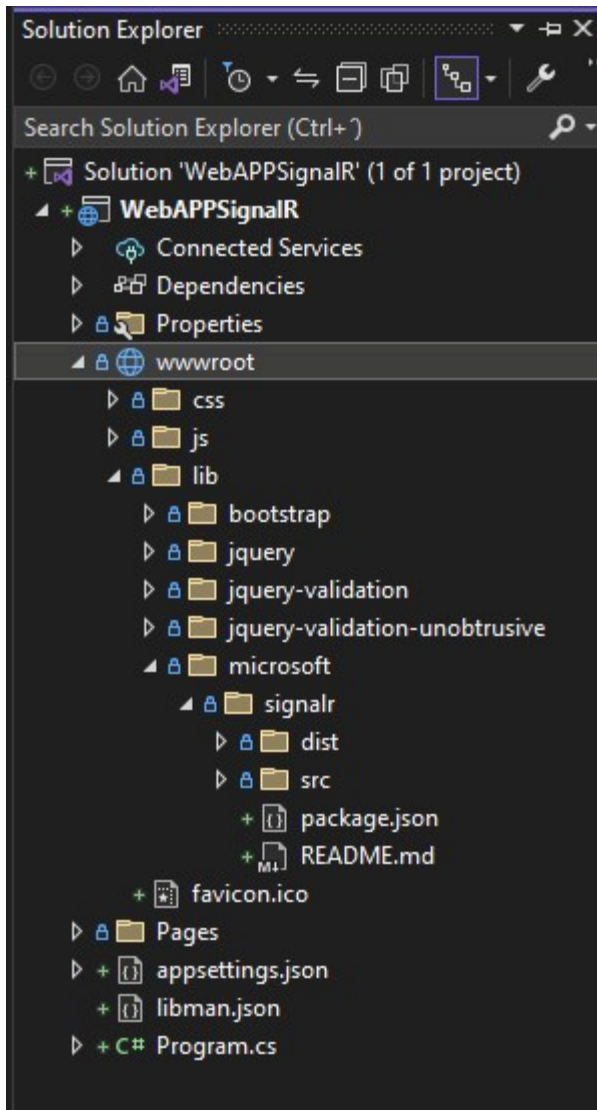


Figure 9.2: Project Solution after adding SignalR client-side library.

Now, let us create the Hub, responsible for communicating among clients.

Add a new class, that must inherit from Hub. The sample `Hub` class is the one as follows:

```
1. public class SampleHub : Hub
2.     {
3.         public async Task
4.         SendMessageToAll(string message)
5.         {
6.             await
7.             Clients.All.SendAsync("ReceiveMessage",
8.             message);
9.         }
10.        public async Task
11.        SendMessageToCaller(string message)
12.        {
13.            await
14.            Clients.Caller.SendAsync("ReceiveMessage",
15.            message);
16.        }
17.        public async Task JoinGroup(string
18.        group)
19.        {
20.            await
21.            Groups.AddToGroupAsync(Context.ConnectionId,
22.            group);
23.        }
24.        public async Task
25.        SendMessageToGroup(string group,
26.        string message)
```



```

23.         {
24.             await
Clients.Group(group).SendAsync("ReceiveMessage",
25.             message);
26.         }
27.         public override async Task
OnConnectedAsync()
28.         {
29.             await
Clients.All.SendAsync("UserConnected",
30.             Context.ConnectionId);
31.             await base.OnConnectedAsync();
32.         }
33.         public override async Task
OnDisconnectedAsync(Exception ex)
34.         {
35.             await
Clients.All.SendAsync("UserDisconnected",
36.             Context.ConnectionId);
37.             await
base.OnDisconnectedAsync(ex);
38.         }
39.     }

```

Now that we have our Hub, we must have the client-side code, this client-side code should communicate with our Hub. The sample client-side code here is named **chat.js** and has the following content:

```

1. "use strict";
2. var connection = new
signalR.HubConnectionBuilder().
3.     withUrl("/
sampleHubRoutePattern").build();
4. connection.on("ReceiveMessage", function
(message) {
5.     var msg = message.replace(/&/g,
"&amp;").replace(/</g, "&lt;");

```

```
6.         .replace(</>/g, "&gt;");
7.         var div =
document.createElement("div");
8.         div.innerHTML = msg + "<hr/>";
9.
document.getElementById("messages").appendChild(
10. });
11. connection.on("UserConnected", function
(connectionId) {
12.     var groupElement =
document.getElementById("group");
13.     var option =
document.createElement("option");
14.     option.text = connectionId;
15.     option.value = connectionId;
16.     groupElement.add(option);
17. });
18. connection.on("UserDisconnected", function
(connectionId) {
19.     var groupElement =
document.getElementById("group");
20.     for (var i = 0; i <
groupElement.length; i++) {
21.         if (groupElement.options[i].value
== connectionId) {
22.             groupElement.remove(i);
23.         }
24.     }
25. });
26. connection.start().catch(function (err) {
27.     return console.error(err.toString());
28. });
29. document.getElementById("sendButton").addEventListener
30.     function (event) {
31.         var message =
document.getElementById("message").value;
```

```

32.     var user =
document.getElementById("userInput").value;
33.     var groupElement =
document.getElementById("group");
34.     var groupValue = groupElement
35.     .options[groupElement.selectedIndex].value;
36.     var finalMessage = `${user} says
${message}`;
37.
38.     if (groupValue === "All" || groupValue
=== "Myself") {
39.         var method = groupValue === "All" ?
"SendMessageToAll"
40.         : "SendMessageToCaller";
41.         connection.invoke(method,
finalMessage)
42.         .catch(function (err) {
43.             return
console.error(err.toString());
44.         });
45.     } else if (groupValue ===
"PrivateGroup") {
46.         connection.invoke("SendMessageToGroup",
"PrivateGroup",
47.         finalMessage).catch(function
(err) {
48.             return
console.error(err.toString());
49.         });
50.     } else {
51.         connection.invoke("SendMessageToUser",
groupValue,
52.         finalMessage).catch(function (err)
{

```

```

53.         return
54.         console.error(err.toString());
55.     });
56.     event.preventDefault();
57. });
58.
59. document.getElementById("joinGroup").addEventListener
60.     function (event) {
61.         connection.invoke("JoinGroup",
62.             "PrivateGroup")
63.             .catch(function (err) {
64.                 return
65.                 console.error(err.toString());
66.             });
67.         event.preventDefault();
68.     });

```

Now, we update the **Index.cshtml** page in order to make use of our client-side code created previously. We have the updated code from the **Index.cshtml** below:

```

1. @page
2. @model IndexModel
3. @{
4.     ViewData[«Title»] = «Home page»;
5. }
6.
7. <div class="container">
8.     <div class="row">&nbsp;</div>
9.
10.    <div class="row">
11.        <div class="col-6">
12.            <input type="button"
13.                id="joinGroup"
14.                value="Join Private
15.                Group" />
16.        </div>

```

```
15.     </div>
16.
17.     <div class="row">
18.
19.         <div class="col-2">Send message
to</div>
20.         <div class="col-4">
21.             <select id="group">
22.                 <option
value="All">Everyone</option>
23.                 <option
value="Myself">Myself</option>
24.                 <option
value="PrivateGroup">Private Group</option>
25.             </select>
26.         </div>
27.     </div>
28.     <div class="row">
29.         <div class="col-2">User</div>
30.         <div class="col-4"><input
type="text" id="userInput" /></div>
31.     </div>
32.     <div class="row">
33.         <div class="col-2">Message</div>
34.         <div class="col-4"><input
type="text" id="message" /></div>
35.     </div>
36.     <div class="row">&nbsp;</div>
37.     <div class="row">
38.         <div class="col-6">
39.             <input type="button"
id="sendButton" value="Send Message"/>
40.         </div>
41.     </div>
42. </div>
43. <div class="row">
```

```

44.     <div class="col-12">
45.         <hr />
46.     </div>
47. </div>
48. <div class="row">
49.     <div class="col-6">
50.         <ul id="messages"></ul>
51.     </div>
52. </div>
53.
54. <script src="~/lib/microsoft/signalr/dist/
    browser/signalr.js"/>
55. <script src="~/lib/microsoft/signalr/dist/
    browser/signalr.min.js"/>
56. <script src="~/js/chat.js"/>

```

At last, we should configure the SignalR in our project startup. To do this, you should have your **Program.cs** as follows:

```

1. using WebAPPSignalR;
2. using Microsoft.AspNetCore.Identity;
3. using Microsoft.EntityFrameworkCore;
4. using WebAPPSignalR.Data;
5. using Microsoft.AspNetCore.SignalR;
6.
7. var builder =
    WebApplication.CreateBuilder(args);
8.
9.
10. // Add services to the container.
11. builder.Services.AddRazorPages();
12. builder.Services.AddSignalR();
13.
14. var app = builder.Build();
15.
16. // Configure the HTTP request pipeline.
17. if (!app.Environment.IsDevelopment())

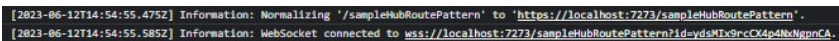
```

```

18. {
19.     app.UseExceptionHandler("/Error");
20.     // The default HSTS value is 30 days.
21.     // You may want to change this for production scenarios,
22.     // see https://aka.ms/aspnetcore-hsts.
23.     app.UseHsts();
24. }
25.
26. app.UseHttpsRedirection();
27. app.UseStaticFiles();
28.
29. app.UseRouting();
30.
31.
32. app.MapRazorPages();
33. app.MapHub<SampleHub>("/sampleHubRoutePattern");
34.
35. app.Run();

```

After this, you can run your project by pushing **F5**, and on the DevTools console, you must see a success message about connecting with SignalR as follows:



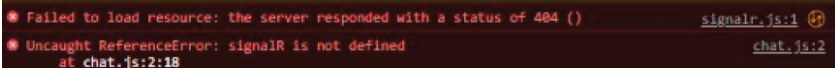
```

[2023-06-12T14:54:55.575Z] Information: Normalizing '/sampleHubRoutePattern' to 'https://localhost:7273/sampleHubRoutePattern'.
[2023-06-12T14:54:55.585Z] Information: WebSocket connected to wss://localhost:7273/sampleHubRoutePattern?id=ydsMTx9rcCX4p4NnNgnCA.

```

Figure 9.3: Visual studio debug console with the success message

The following error message is a common error when not being able to connect with SignalR:



```

✖ Failed to load resource: the server responded with a status of 404 () signalr.js:1
✖ Uncaught ReferenceError: signalR is not defined chat.js:2
  at chat.js:2:18

```

Figure 9.4: Chrome DevTools console with the error message

To fix the previous error, you should open the command prompt and execute both commands:

- **dotnet dev-certs https --clean**

- `dotnet dev-certs https --trust`

The first command is responsible to remove all HTTPS development certificates, while the second command is responsible to trust on the generated certificate.

Authorization and authentication

In this example, we are using the identity engine managed by a SQL Server instance, accounts and roles are saved in the database.

Required Nuget packages:

- `Microsoft.AspNetCore.Identity.EntityFrameworkCore`
- `Microsoft.AspNetCore.Identity.UI`
- `Microsoft.EntityFrameworkCore`
- `Microsoft.EntityFrameworkCore.SqlServer`
- `Microsoft.EntityFrameworkCore.Tools`

Add a new scaffolded item of type identity. You can find it in the identity sub-menu on the left:

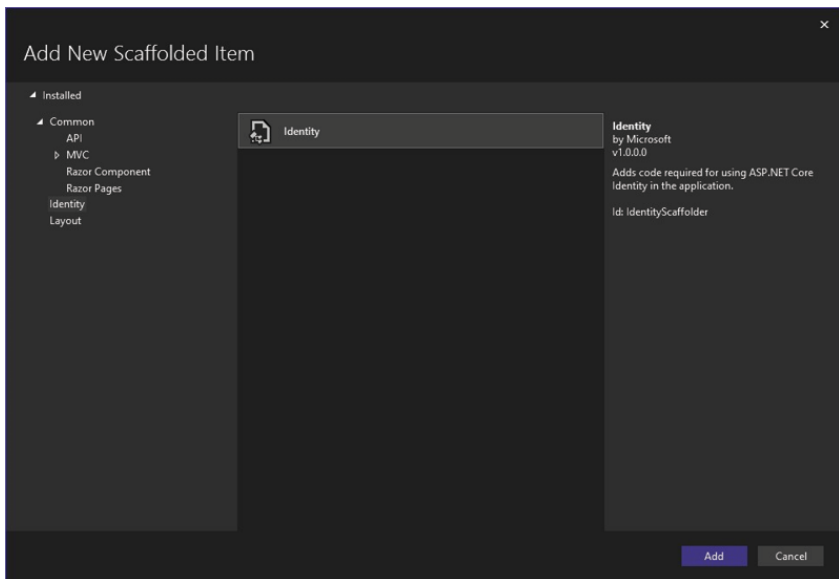


Figure 9.5: Adding an Identity Scaffolded item.

Check the **Override all files** option and set your **DbContext** class:

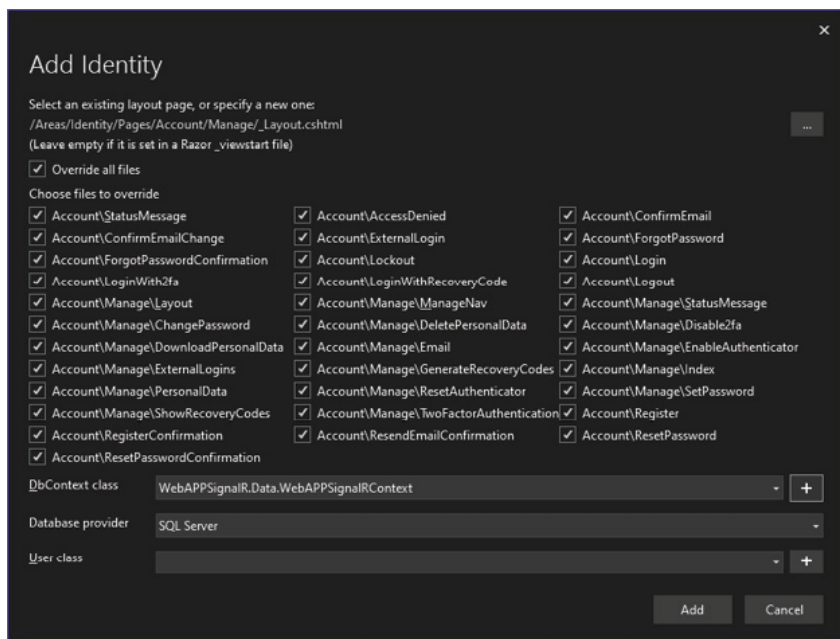


Figure 9.6: Setting up Identity scaffolded items.

We have the following classes created as a result:

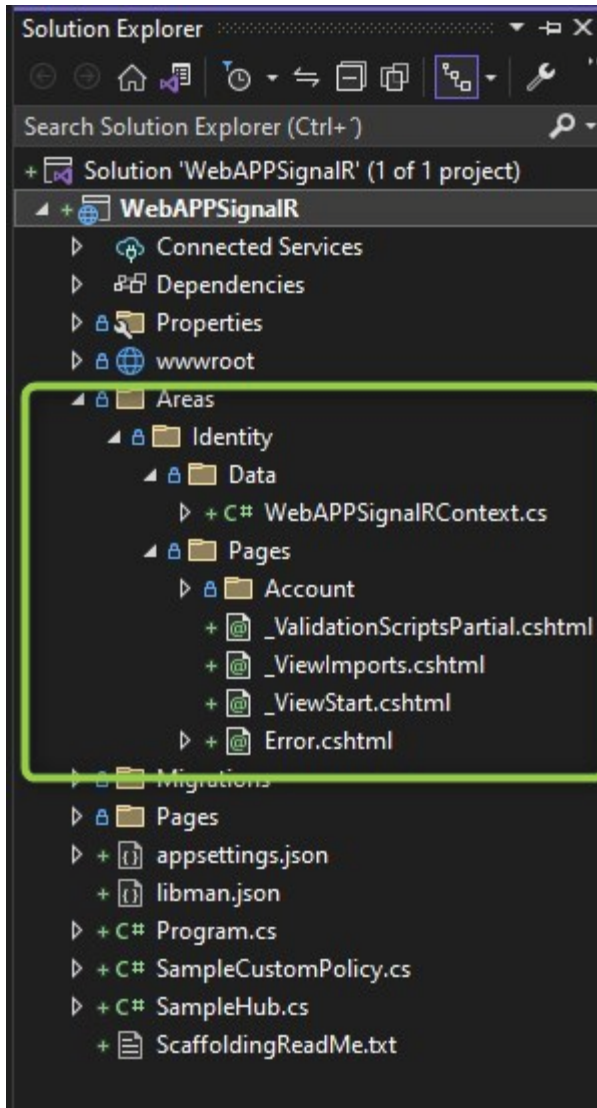


Figure 9.7: Solution Explorer with the items created after the Identity's scaffolding process.

Now, the **Program.cs** has to be updated in order to connect to the database, as follows:

1. `using WebAPPSignalR;`
2. `using Microsoft.AspNetCore.Identity;`
3. `using Microsoft.EntityFrameworkCore;`

```

4. using WebAPPSignalR.Data;
5.
6. var builder =
    WebApplication.CreateBuilder(args);
7.
8. builder.Services.AddDbContext<WebAPPSignalRContext>
    =>
9.     options.UseSqlServer("Data Source=
10. DESKTOP-H20012E;Initial
    Catalog=SampleSignalRIdentityDb;
11. Integrated Security=True;Connect
    Timeout=30;Encrypt=False;
12. Trust Server Certificate=False;
13. Application Intent=ReadWrite;Multi Subnet
    Failover=False»));
14.
15. builder.Services.AddDefaultIdentity<IdentityUser>
    =>
16.     options.SignIn.RequireConfirmedAccount =
    true);
17.
18.     .AddEntityFrameworkStores<WebAPPSignalRContext>());
19.
20. // Add services to the container.
21. builder.Services.AddRazorPages();
22. builder.Services.AddSignalR();
23.
24. var app = builder.Build();
25.
26. // Configure the HTTP request pipeline.
27. if (!app.Environment.IsDevelopment())
28. {
29.     app.UseExceptionHandler("/Error");
30.     // The default HSTS value is 30 days.
31.     // You may want to change this for production scenarios
    // see https://aka.ms/aspnetcore-hsts.

```

```
32.     app.UseHsts();
33. }
34.
35. app.UseHttpsRedirection();
36. app.UseStaticFiles();
37.
38. app.UseRouting();
39.
40. app.UseAuthentication();
41. app.UseAuthorization();
42.
43. app.MapRazorPages();
44. app.MapHub<SampleHub>("/
    sampleHubRoutePattern");
45.
46. app.Run();
```

After creating all the objects and setting up our project startup, we need to run the commands in the Package Manager Console window. These commands are responsible for creating and setting up our identity engine created and set up in the database.

- Add migration
- Update-database

After successfully executing the commands, those are the tables created in the database. Those tables are responsible for managing user accounts, authentication, and authorization. You may see the tables that were automatically created in the image below:

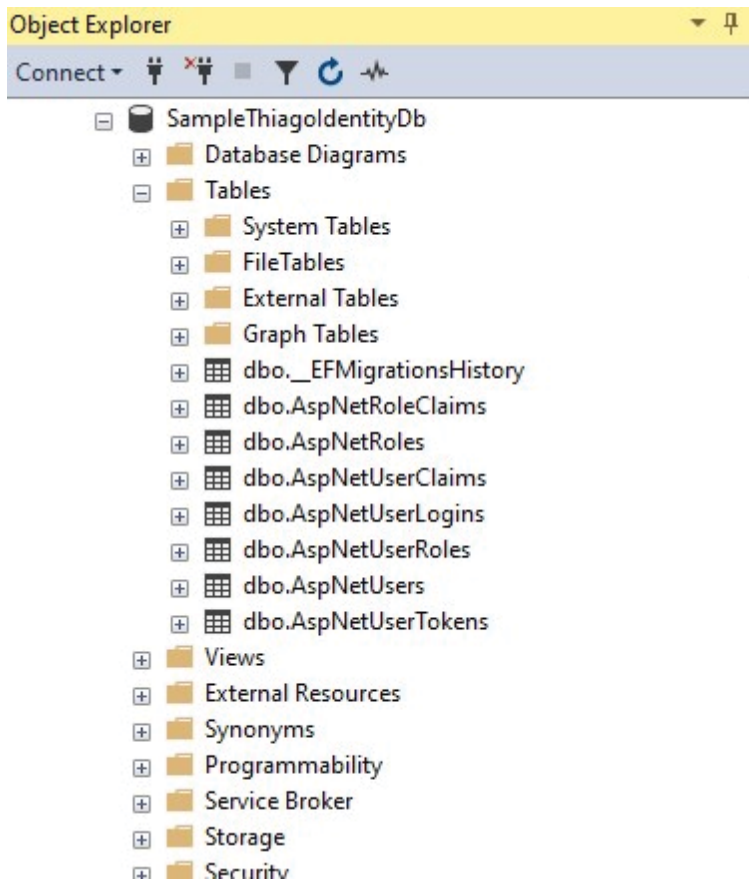


Figure 9.8: SQL Server with the tables created

Now, we need to update our project objects to implement authentication and authorization. The following changes are required to make use of the authorization and authentication:

Our **Hub** class had to be added the authorize attribute:

```

1. [Authorize]
2. public class SampleHub : Hub
3. {
4.     public async Task
       SendMessageToAll(string message)
5.     {
6.         string userName =
           Context.User.Identity.Name;

```

```

7.         await
Clients.All.SendAsync("ReceiveMessage",
8.             $"User {userName} says « +
message);
9.     }
10.    public async Task
SendMessageToCaller(string message)
11.    {
12.        string userName =
Context.User.Identity.Name;
13.        await
Clients.Caller.SendAsync("ReceiveMessage",
14.            $"User {userName} says « +
message);
15.    }
16.    public async Task
SendMessageToUser(string connectionId,
17.        string message)
18.    {
19.        string userName =
Context.User.Identity.Name;
20.        await
Clients.Client(connectionId)
21.            .SendAsync("ReceiveMessage",
22.                $"User {userName} says « +
message);
23.    }
24.    public async Task JoinGroup(string
group)
25.    {
26.        await
Groups.AddToGroupAsync(Context.ConnectionId,
group);
27.    }
28.    public async Task
SendMessageToGroup(string group,
29.        string message)

```

```

30.         {
31.             string userName =
Context.User.Identity.Name;
32.             await
Clients.Group(group).SendAsync("ReceiveMessage",
33.                 $"User {userName} says «+
message»);
34.         }
35.         public override async Task
OnConnectedAsync()
36.         {
37.             await
Clients.All.SendAsync("UserConnected",
38.                 Context.ConnectionId);
39.
40.             await base.OnConnectedAsync();
41.         }
42.         public override async Task
OnDisconnectedAsync(Exception ex)
43.         {
44.             await
Clients.All.SendAsync("UserDisconnected",
45.                 Context.ConnectionId);
46.             await
base.OnDisconnectedAsync(ex);
47.         }
48.     }

```

We are using the identity from the logged in user, so we do not need a textbox for the user to input his username anymore.

Therefore, we removed the username input from the

Index.cshtml:

```

1. @page
2. @model IndexModel
3. @{
4.     ViewData[«Title»] = «Home page»;
5. }
6.

```

```
7. <div class="container">
8.     <div class="row">&nbsp;</div>
9.
10.    <div class="row">
11.        <div class="col-6">
12.            <input type="button"
13.                id="joinGroup"
14.                value="Join Private
15.                Group" />
16.        </div>
17.    </div>
18.
19.    <div class="row">
20.        <div class="col-2">Send message
21.        to</div>
22.        <div class="col-4">
23.            <select id="group">
24.                <option
25.                    value="All">Everyone</option>
26.                <option
27.                    value="Myself">Myself</option>
28.                <option
29.                    value="PrivateGroup">Private Group</option>
30.            </select>
31.        </div>
32.    </div>
33.    <div class="row">
34.        <div class="col-2">Message</div>
35.        <div class="col-4"><input
36.            type="text" id="message" /></div>
37.    </div>
38.    <div class="row">&nbsp;</div>
39.    <div class="row">
40.        <div class="col-6">
41.            <input type="button"
```



```

        id="sendButton" value="Send Message"/>
36.         </div>
37.     </div>
38. </div>
39. <div class="row">
40.     <div class="col-12">
41.         <hr />
42.     </div>
43. </div>
44. <div class="row">
45.     <div class="col-6">
46.         <ul id="messages"></ul>
47.     </div>
48. </div>
49.
50. <script src="~/lib/microsoft/signalr/dist/
    browser/signalr.js"/>
51. <script src="~/lib/microsoft/signalr/dist/
    browser/signalr.min.js"/>
52. <script src="~/js/chat.js"></script>

```

The **Chat.js** was also updated to reflect the changes in the previous page:

```

1. "use strict";
2. var connection = new
    signalR.HubConnectionBuilder()
3.         .withUrl("/")
    sampleHubRoutePattern").build();
4. connection.on("ReceiveMessage", function
    (message) {
5.     var msg = message.replace(/&/g,
    "&amp;").replace(/</g, "&lt;");
6.         .replace(/>/g, "&gt;");
7.     var div =
    document.createElement("div");
8.     div.innerHTML = msg + "<hr/>";
9.

```

```

        document.getElementById("messages").appendChild(
10. });
11. connection.on("UserConnected", function
    (connectionId) {
12.     var groupElement =
        document.getElementById("group");
13.     var option =
        document.createElement("option");
14.     option.text = connectionId;
15.     option.value = connectionId;
16.     groupElement.add(option);
17. });
18. connection.on("UserDisconnected", function
    (connectionId) {
19.     var groupElement =
        document.getElementById("group");
20.     for (var i = 0; i <
        groupElement.length; i++) {
21.         if (groupElement.options[i].value
            == connectionId) {
22.             groupElement.remove(i);
23.         }
24.     }
25. });
26. connection.start().catch(function (err) {
27.     return console.error(err.toString());
28. });
29. document.getElementById("sendButton")
30.     .addEventListener("click", function
        (event) {
31.         var message =
            document.getElementById("message").value;
32.         var groupElement =
            document.getElementById("group");
33.         var groupValue = groupElement
34.             .options[groupElement.selectedIndex].value;

```

```

35.
36.
37.     if (groupValue === "All" || groupValue
=== "Myself") {
38.         var method = groupValue === "All" ?
"SendMessageToAll"
39.             : "SendMessageToCaller";
40.         connection.invoke(method,
message).catch(function (err) {
41.             return
console.error(err.toString());
42.         });
43.     } else if (groupValue ===
"PrivateGroup") {
44.         connection.invoke("SendMessageToGroup",
"PrivateGroup",
45.             message).catch(function (err) {
46.                 return
console.error(err.toString());
47.             });
48.     } else {
49.         connection.invoke("SendMessageToUser",
groupValue, message)
50.             .catch(function (err) {
51.                 return
console.error(err.toString());
52.             });
53.     }
54.
55.     event.preventDefault();
56. });
57. document.getElementById("joinGroup").addEventListener
58.     function (event) {
59.         connection.invoke("JoinGroup",
"PrivateGroup")

```

```
60.         .catch(function (err) {  
61.             return  
        console.error(err.toString());  
62.         });  
63.         event.preventDefault();  
64.     });
```

Now that we have our database and project ready let us register a new user. Using your browser, go to **/Identity/Account/ Register**:

WebAPPSignalR Home Privacy

Register

Create a new account.

Email

Password

Confirm Password

Register

Figure 9.9: The Web APP registration form

After successfully completing the registration process and going to the chat, this is the output:

WebAPPSignalR Home Privacy

Join Private Group

Send message to

Message

Everyone

hello

Send Message

User [redacted]@gmail.com says hello

Figure 9.10: My user sending a “hello” message to everyone.

Custom authorization policy

The new custom authorization policy will allow me to send messages to everyone but not allow me to send messages to myself.

For this, we have to create a new class to implement the custom policy implementation as you can see in the following code:

```
1. public class SampleCustomPolicy :
2.     AuthorizationHandler<SampleCustomPolicy,
3.         HubInvocationContext>,
4.         IAuthorizationRequirement
5.     {
6.         protected override Task
7.             HandleRequirementAsync (
8.                 AuthorizationHandlerContext context,
```

```

resource)
9.     {
10.         if (context.User.Identity != null
    &&
11.             !
    string.IsNullOrEmpty(context.User.Identity.Name)
    &&
12.             IsUserAllowedToDoThis(resource.HubMethodName,
13. context.User.Identity.Name))
14.         {
15.             context.Succeed(requirement);
16.
17.         }
18.         return Task.CompletedTask;
19.     }
20.     private bool
    IsUserAllowedToDoThis(string hubMethodName,
21. string currentUsername)
22.     {
23.         return !
    (currentUsername.EndsWith("@gmail.com") &&
24.         hubMethodName.Equals("SendMessageToCaller",
    StringComparison.OrdinalIgnoreCase));
25.     }
26. }

```

We need to update the **program.cs** so it knows that we have a custom authorization policy in place:

```

1. using WebAPPSignalR;
2. using Microsoft.AspNetCore.Identity;
3. using Microsoft.EntityFrameworkCore;
4. using WebAPPSignalR.Data;
5.

```

```
6. var builder =
    WebApplication.CreateBuilder(args);
7.
8. builder.Services.AddDbContext<WebAPPSignalRContext>
    =>
9.     options.UseSqlServer("Data
    Source=DESKTOP-H20012E;
10. Initial
    Catalog=SampleSignalRIdentityDb;Integrated
    Security=True;
11. Connect Timeout=30;Encrypt=False;Trust
    Server Certificate=False;
12. Application Intent=ReadWrite;Multi Subnet
    Failover=False»));
13.
14. builder.Services.AddDefaultIdentity<IdentityUser>
    =>
15.     options.SignIn.RequireConfirmedAccount =
    true)
16.
    .AddEntityFrameworkStores<WebAPPSignalRContext>();
17.
18. builder.Services.AddAuthorization(options
    =>
19. {
20.     options.AddPolicy("SamplePolicyName",
    policy =>
21.     {
22.         policy.Requirements.Add(new
    SampleCustomPolicy());
23.     });
24. });
25.
26. // Add services to the container.
27. builder.Services.AddRazorPages();
28. builder.Services.AddSignalR();
```

```

29.
30. var app = builder.Build();
31.
32. // Configure the HTTP request pipeline.
33. if (!app.Environment.IsDevelopment())
34. {
35.     app.UseExceptionHandler("/Error");
36.     // The default HSTS value is 30 days.
37.     //You may want to change this for production scenarios,
38.     //see https://aka.ms/aspnetcore-hsts.
39.     app.UseHsts();
40. }
41.
42. app.UseHttpsRedirection();
43. app.UseStaticFiles();
44.
45. app.UseRouting();
46.
47. app.UseAuthentication();
48. app.UseAuthorization();
49.
50. app.MapRazorPages();
51. app.MapHub<SampleHub>("/
    sampleHubRoutePattern");
52.
53. app.Run();

```

Also, we have to update our Hub to have this custom policy enforced. Update the **samplehub.cs**:

```

1. using Microsoft.AspNetCore.Authorization;
2. using Microsoft.AspNetCore.SignalR;
3.
4. namespace WebAPPSignalR
5. {
6.     [Authorize]
7.     public class SampleHub : Hub
8.     {

```



```

9.         [Authorize("SamplePolicyName")]
10.         public async Task
SendMessageToAll(string message)
11.         {
12.             string userName =
Context.User.Identity.Name;
13.             await
Clients.All.SendAsync("ReceiveMessage",
14.                 $"User {userName} says
« + message);
15.         }
16.         [Authorize("SamplePolicyName")]
17.         public async Task
SendMessageToCaller(string message)
18.         {
19.             string userName =
Context.User.Identity.Name;
20.             await
Clients.Caller.SendAsync("ReceiveMessage",
21.                 $"User {userName}
says « + message);
22.         }
23.         public async Task
SendMessageToUser(string connectionId,
24.                 string message)
25.         {
26.             string userName =
Context.User.Identity.Name;
27.             await
Clients.Client(connectionId)
28.                 .SendAsync("ReceiveMessage",
$"User {userName} says «
29.                 + message);
30.         }
31.         public async Task JoinGroup(string
group)
32.         {

```

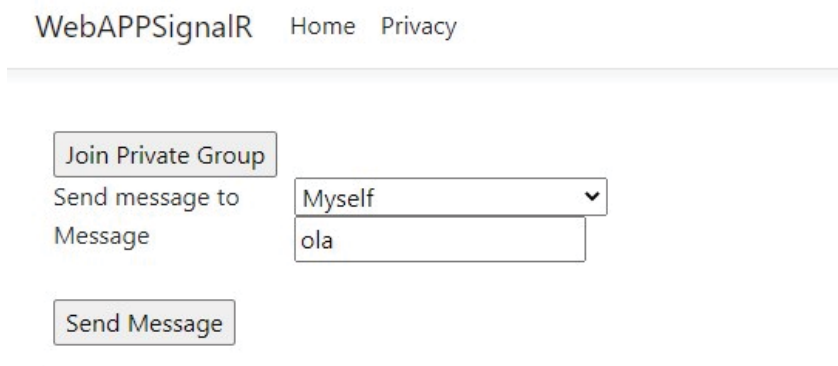
```

33.         await
Groups.AddToGroupAsync (Context.ConnectionId,
group);
34.     }
35.     public async Task
SendMessageToGroup(string group,
36.                   string message)
37.     {
38.         string userName =
Context.User.Identity.Name;
39.         await
Clients.Group(group).SendAsync ("ReceiveMessage",
40.                                $"User {userName} says
«+ message);
41.     }
42.     public override async Task
OnConnectedAsync()
43.     {
44.         await
Clients.All.SendAsync ("UserConnected",
45.                        Context.ConnectionId);
46.
47.         await base.OnConnectedAsync();
48.     }
49.     public override async Task
OnDisconnectedAsync(Exception ex)
50.     {
51.         await
Clients.All.SendAsync ("UserDisconnected",
52.                        Context.ConnectionId);
53.         await
base.OnDisconnectedAsync (ex);
54.     }
55. }
56. }

```

We have this output after running our project and trying to send a

message to myself:



The screenshot shows a web application interface for WebAPPSignalR. At the top, there are navigation links for 'Home' and 'Privacy'. Below these, there is a 'Join Private Group' button. Underneath, the text 'Send message to' is followed by a dropdown menu currently showing 'Myself'. Below that, the text 'Message' is followed by a text input field containing the word 'ola'. At the bottom of this section is a 'Send Message' button.

Figure 9.11: My user sending a message to myself

Conclusion

Throughout this chapter, we have explored the exciting world of real-time communication and how SignalR, together with ASP.NET, empowers you to build engaging and interactive applications.

We began by understanding the importance of real-time communication in today's digital landscape and how it enhances user experiences. SignalR emerged as a powerful framework that simplifies the process of adding real-time capabilities to your applications, enabling seamless bidirectional communication between the server and the client.

We dived into the configuration aspects of SignalR, ensuring that you have a solid foundation to work with. By understanding the various configuration options and settings, you can customize SignalR to suit your specific application requirements, achieving optimal performance and scalability.

Authentication and authorization were essential considerations in real-time communication scenarios, and we explored how to secure SignalR connections. By implementing robust authentication mechanisms, you can control access to your real-time features, ensuring that only authorized users can interact with your application.

Furthermore, we delved into the Streaming Hub feature of SignalR, which enables real-time streaming scenarios. You learned how to implement streaming hubs and optimize performance for efficient streaming of large data sets. This knowledge equips you to build applications that deliver continuous streams of data to clients, enriching the user experience with real-time updates.

To reinforce your understanding, we concluded the chapter with a comprehensive case study. Step by step, you followed along and witnessed the practical implementation of SignalR and ASP.NET in a real-world scenario. This case study provided a hands-on experience, allowing you to apply the concepts and techniques covered throughout the chapter in a practical context.

With the knowledge gained in this chapter, you are now equipped to leverage the power of SignalR and ASP.NET to create dynamic, real-time applications for cloud, web, and desktop platforms. Real-time communication opens up a realm of possibilities for engaging user experiences, collaborative environments, and data-driven interactions.

As you continue your journey in application development, remember to explore and experiment with the concepts covered in this chapter. Stay updated with the latest advancements in SignalR and ASP.NET, as new features and improvements are constantly being introduced. By embracing real-time communication, you can create applications that captivate and delight your users.

In the upcoming chapter, we will discuss implementing microservices with Web APIs. This pivotal topic addresses the contemporary approach of building scalable and agile systems through microservices architecture. We delve into the intricacies of designing and deploying microservices, with a specific focus on Web APIs as the foundation for seamless communication between services. Readers will explore the principles, benefits, and practical considerations of microservices, gaining insights into how this architectural paradigm enhances flexibility, scalability, and overall system resilience. Join us as we navigate the landscape of microservices, unraveling the key concepts and practical strategies to effectively implement them using Web APIs.

Best of luck as you continue building innovative applications with SignalR and ASP.NET!

CHAPTER 10

Implementing MicroServices with Web APIs

Introduction

This chapter provides an in-depth guide to building scalable and resilient microservices using Web APIs. The chapter focuses on scaling, which is one of the critical challenges of building microservices-based applications.

The chapter begins by providing an overview of microservices architecture and the benefits of using Web APIs for building microservices. It then explains how to design a scalable microservices architecture, including topics like service discovery, load balancing, and fault tolerance.

The chapter then dives into the various scaling techniques that can be used for microservices-based applications, including horizontal scaling, vertical scaling, and auto-scaling. It provides step-by-step instructions for implementing each scaling technique, along with best practices and common pitfalls to avoid.

To provide a practical case study, the chapter walks through building a simple but functional microservices-based application that incorporates all the scaling techniques discussed in the chapter. It demonstrates how to design a scalable microservices architecture, how to implement scaling technique and use Web APIs to communicate between microservices.

Structure

This chapter covers the following topics:

- Implementing MicroServices with WebAPIs
- Asynchronous communication
- RabbitMQ
- MicroServices scalability
- Horizontal scalability
- Vertical scalability
- Orchestrators
- Most used architectural patterns
- Backend for frontend
- Command Query Responsibility Segregation
- Domain Driven Design
- Case study

Objectives

This chapter equips you with fundamental insights into microservices, including an overview and asynchronous communication. It delves into scaling, covering horizontal and vertical scaling, along with considerations, benefits, and nuanced management. Best practices for scaling are outlined, accompanied by an introduction to key microservices architectural patterns clarifying their roles in scalability and resilience. A detailed case study guides you through implementing a microservices-based application, applying the BFF design pattern and exploring both synchronous and asynchronous communication. This chapter provides both theoretical insights and practical skills for

constructing scalable and resilient microservices with Web APIs, spanning scaling strategies, architectural patterns, and real-world application.

Implementing MicroServices with WebAPIs

Microservices architecture is an approach to software development that emphasizes the decomposition of a complex application into smaller, independent services. Each service is designed to perform a specific business capability and can be developed, deployed, and scaled independently. These services communicate with each other through well-defined APIs, enabling them to work together seamlessly. Microservices offer several advantages, including improved scalability, flexibility, and resilience. They allow teams to work in parallel, use different technologies for each service, and easily replace or upgrade individual components. However, managing the complexity of distributed systems and ensuring effective communication between services are challenges that need to be addressed. With proper design and implementation, microservices can enable organizations to build highly modular and adaptable systems that can rapidly respond to changing business requirements.

Asynchronous communication

Asynchronous communication is extremely important when working with microservices architectures, making it a must-have when trying to scale those microservices. With asynchronous communication we may communicate between microservices, or we can process heavier workloads in the background while providing immediate response to the original request. This allows us to have a better responsiveness.

By processing heavier workloads in the background, we can have a higher control over the workload being processed among the microservices, this approach offers several benefits:

- **Increased responsiveness:** A microservice does not need to be stuck waiting for a response, it can continually process other tasks in the background while waiting for a response from an asynchronous operation.

- **Improved resilience:** If a microservice fails or gets unavailable it does not affect other microservices, because they can continue operating independently and handle messages asynchronously. When the microservice is fixed and goes back online it will process all the requests in the pipeline.
- **Scalability and load balancing:** Asynchronous communication facilitates horizontal scaling of microservices by sharing the exchanged messages among different processors. This approach allows us to scale and load balance the workload among all the instances of this processor.
- **Loose coupling and flexibility:** Asynchronous communication promotes loose coupling between microservices. They can evolve independently without impacting others as long as the message contracts remain compatible.

To implement asynchronous communication, microservices publish messages to designated channels or queues. Then, we have other microservices acting as consumers of those messages and processing the request asynchronously.

Let us explore how to handle asynchronous communication using RabbitMQ.

RabbitMQ

RabbitMQ is an open-source message broker that serves as a reliable intermediary for exchanging messages between publishers and listeners. It implements the message queue protocol, enabling seamless communication across various channels and routes. With RabbitMQ, publishers can send messages to specific queues, and listeners can consume those messages when they are ready. This decoupling of message producers and consumers allows for flexible and scalable communication between different components of a system. RabbitMQ provides robust features such as message persistence, message routing, and support for multiple messaging patterns. It is widely used in distributed systems, microservices architectures, and asynchronous communication scenarios where reliable message exchange is essential.

Among RabbitMQ's main features and keywords, several important ones are highlighted below, along with a brief explanation of each:

- **Message broker:** RabbitMQ acts as a message broker, facilitating the exchange of messages between different components of a system. It ensures reliable delivery and efficient routing of messages.
- **Message queue protocol:** RabbitMQ implements a message queue protocol, which defines the rules and formats for exchanging messages. This protocol allows publishers to send messages to specific queues and listeners to consume messages from those queues.
- **Message:** Messages play a vital role as they traverse channels from publishers to listeners. These messages can contain a wide range of information, ranging from basic text to intricate serialized objects. They act as carriers of data, enabling seamless communication and exchange of information between different components within the RabbitMQ system. Whether it is transmitting straightforward textual content or intricate serialized objects, messages facilitate the efficient flow of data throughout the RabbitMQ infrastructure.
- **Channel:** Serves as a logical communication line between publishers and listeners. It operates within an established connection and facilitates the transfer of messages using various protocols. By utilizing channels, publishers can efficiently send messages to listeners, enabling seamless communication and data exchange within the RabbitMQ system. Channels enhance the flexibility and versatility of message transfer, providing a reliable and efficient means of communication between different components.
- **Queue:** A RabbitMQ queue operates on the **First-In-First-Out (FIFO)** principle, providing a mechanism for storing messages between publishers and listeners. Messages are stored in the queue in the order they are received, and they are retrieved and processed by listeners in the same order. This ensures that the messages maintain their original sequence and are processed in a fair and consistent manner.
- **Connection:** Forms the foundation for communication between the server and the client. It establishes the link between the two by leveraging protocols. Once the connection is established, it enables the creation and operation of channels, which serve as logical communication pathways within the RabbitMQ system. The connection acts as a bridge, facilitating the exchange of messages and data between the

server and the client, enabling seamless communication and collaboration.

- **Publisher-subscriber model:** RabbitMQ supports the publisher-subscriber model, where publishers send messages to specific queues, and subscribers (or listeners) consume those messages as needed. This decoupling enables asynchronous communication and flexible scaling.
- **Consumer:** A consumer is a component connected to a RabbitMQ client that listens to specific queues for message consumption. It retrieves and processes messages published on those queues.
- **Publisher:** A publisher is a component connected to a RabbitMQ client that sends messages to a specific queue for publishing. It is responsible for producing and delivering messages to the designated queue for further processing or distribution.
- **Notification:** Notifications in the context of microservices are crucial for monitoring the health of services and can be customized to trigger alerts based on specific metrics. These notifications serve as a mechanism to keep track of the performance, availability, and overall well-being of microservices. By defining thresholds and conditions, notifications can be set up to proactively detect and report any anomalies or deviations from expected behavior. This enables timely response and intervention, allowing teams to address potential issues and maintain the smooth operation of their services. Customizable notifications provide flexibility in tailoring alerts to the specific needs and requirements of the system, ensuring that the right stakeholders are promptly notified when critical events or metrics are triggered.
- **Dead letter:** Dead letters are utilized to store messages that were unable to be consumed by their intended listeners. This can occur if the message is rejected by the listeners, the queue becomes full, or the message reaches its expiration time. Dead letter queues serve as a fallback mechanism, allowing these unprocessed messages to be redirected and stored for further analysis or alternative processing. By leveraging dead letters, RabbitMQ provides a way to handle and manage messages that could not be consumed in their original context, ensuring message reliability and fault tolerance within the system.
- **Route:** RabbitMQ routes play a crucial role in ensuring the

targeted delivery of messages to their respective queues based on routing keys and exchanges. These routes act as a mechanism for directing messages to their intended recipients, enabling precise and efficient message distribution within the RabbitMQ system. By evaluating the routing key associated with each message, RabbitMQ routes determine the appropriate destination queue to which the message should be delivered. This ensures that messages reach the specific consumers or services that are interested in processing them, facilitating effective communication and message handling in a structured and organized manner.

- **Virtual host:** In comparison with a SQL Server database, a RabbitMQ virtual host can be seen as an equivalent to a SQL Server database. Just like a database in SQL Server, a virtual host in RabbitMQ is a self-contained environment with its own set of settings and configurations. Each virtual host operates independently of others, having its own channels, bindings, protocols, users, and other relevant attributes. It provides a logical separation of resources, allowing different applications or services to operate in isolation within their dedicated virtual host. This segregation ensures that the settings and entities within one virtual host do not interfere with or affect those in other virtual hosts, providing a level of autonomy and control over the messaging environment.
- **Exchange:** In RabbitMQ, exchanges play a critical role in routing messages to their respective queues based on their attributes. An exchange acts as a routing agent, receiving messages from publishers and determining their destination queues. The routing decision is made by evaluating attributes such as the message's routing key, headers, or other specified criteria. The exchange then forwards the message to the appropriate queues that match the defined routing rules. This mechanism enables precise and targeted message delivery, ensuring that messages reach the queues that are specifically interested in consuming them. By leveraging exchanges, RabbitMQ provides a flexible and configurable routing mechanism that supports various message distribution patterns and facilitates efficient communication between publishers and consumers. RabbitMQ provides several types of exchanges:
 - **Direct exchange:** This exchange delivers messages to queues based on an exact match between the routing key of

the message and the routing key specified in the binding of the queue.

- **Topic exchange:** Messages sent to a topic exchange are routed to queues based on patterns defined by the routing key. Wildcard characters such as "*" (matches a single word) and "#" (matches zero or more words) allow for flexible routing based on specific patterns.
- **Fanout exchange:** Fanout exchanges broadcast messages to all the queues that are bound to them. The routing key is ignored, and the message is distributed to all the queues.
- **Headers exchange:** Headers exchanges use message headers instead of the routing key for routing decisions. The headers are matched against those specified in the bindings to determine the appropriate destination queues.
- **Bindings:** A RabbitMQ binding links a queue to an exchange, defining the relationship between exchanges and queues. They determine how messages are routed from an exchange to specific queue(s) based on routing rules.

The main benefits of RabbitMQ usage:

- **Multi-platform communication:** RabbitMQ supports message serialization and deserialization in common languages like JSON, enabling seamless communication between different platforms and technologies.
- **Asynchronous operations:** RabbitMQ allows for asynchronous messaging, ensuring that services are not locked or blocked while waiting for a response. This enables efficient and non-blocking communication between components.
- **Open-source and community-driven:** RabbitMQ is an open-source message broker with a vibrant community actively working on its development and improvement. This ensures continuous enhancements, bug fixes, and the availability of extensive resources and support.
- **Multi-language support:** RabbitMQ offers compatibility with a wide range of programming languages, allowing developers to use their preferred language for message production and consumption. This flexibility promotes language diversity and enables teams to work with their preferred tech stack.
- **Multi-protocol support:** RabbitMQ supports multiple protocols for exchanging messages. It provides compatibility

with popular messaging protocols like **Advanced Message Queuing Protocol (AMQP)**, **Message Queuing Telemetry Transport (MQTT)**, and more. This versatility allows for seamless integration with diverse systems and technologies.

- **Reliability and fault-tolerance:** RabbitMQ ensures reliable message delivery by providing features such as message persistence, delivery acknowledgments, and durable queues. It also supports fault-tolerant setups like clustering and mirrored queues, replicating messages across nodes for high availability and data redundancy.
- **Scalability:** RabbitMQ is designed to handle high message throughput and can scale horizontally by adding more nodes to distribute the message processing load. This allows applications to accommodate increasing workloads and handle peak traffic efficiently.
- **Flexible routing and messaging patterns:** RabbitMQ offers various exchange types and routing mechanisms, enabling flexible message routing and supporting different messaging patterns such as publish/subscribe, request/reply, and topic-based filtering. This flexibility allows for the implementation of complex communication patterns in distributed systems.
- **Dead-letter queues:** RabbitMQ provides dead-letter queues, which capture messages that cannot be processed successfully. This feature allows for proper handling and analysis of failed messages, aiding in troubleshooting and debugging of the messaging system.
- **Management and monitoring:** RabbitMQ provides a user-friendly management interface and comprehensive monitoring capabilities. These tools allow administrators to monitor queues, connections, message rates, and other metrics, helping them gain insights into system performance, troubleshoot issues, and perform effective resource management.
- **Extensibility and integration:** RabbitMQ offers a plugin system that allows its functionality to be extended with additional features and protocols. It integrates well with other systems and frameworks, making it compatible with a wide range of tools and technologies commonly used in modern software development.

By leveraging these benefits, RabbitMQ empowers developers to build robust and flexible messaging solutions that facilitate efficient

communication between different components and systems.

MicroServices scalability

Scalability is a critical aspect of building microservices-based applications that can handle increased workloads and adapt to changing demands. In this section, we will delve into the topic of microservices scalability, exploring three key strategies: **Horizontal scalability**, **vertical scalability**, and **orchestrators**.

Microservices scalability refers to the ability to efficiently and effectively handle growing demands by adding resources or redistributing workloads across the system. It plays a pivotal role in ensuring that microservices can handle increased traffic, maintain optimal performance, and provide a seamless user experience.

Horizontal scalability involves scaling out the application horizontally by adding more instances of microservices to the system. This approach allows for distributing the workload across multiple instances, enabling improved performance, higher availability, and easier load balancing.

Vertical scalability, on the other hand, focuses on scaling up individual microservices by increasing the resources allocated to them. This can involve upgrading the hardware, increasing memory capacity, or enhancing processing power. Vertical scalability is particularly useful when a microservice requires more resources to handle specific tasks efficiently.

In addition to horizontal and vertical scalability, we will also explore the role of Orchestrators in managing and coordinating the scaling process. Orchestrators, such as Kubernetes or Docker Swarm, provide containerization and orchestration capabilities, allowing for efficient deployment, scaling, and management of microservices across a cluster of machines.

By understanding and implementing these scalability strategies and utilizing orchestrators effectively, microservices-based applications can achieve the necessary flexibility, performance, and resilience to adapt to varying workloads and ensure a seamless user experience.

Horizontal scalability

Horizontal scalability, also known as **scaling out**, is a strategy for increasing the capacity of a microservices-based application by adding more instances of microservices to the system. Instead of upgrading individual microservices, horizontal scalability focuses on distributing the workload across multiple instances, allowing for improved performance, higher availability, and easier load balancing. You can see how horizontal scalability works in the figure below:

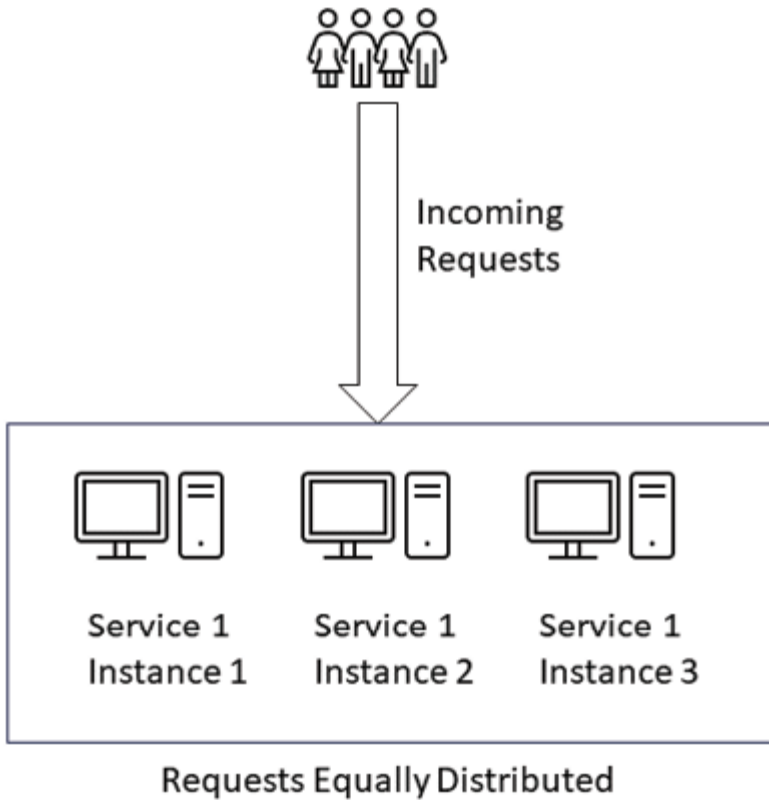


Figure 10.1: Visual explanation of how the Horizontal Scalability works.

When horizontally scaling a microservice, you replicate it across multiple servers or virtual machines, creating a cluster of instances. Each instance is independent and can handle a portion of the incoming requests or workload. This distribution of workload ensures that the overall system can handle increased traffic and

provide a seamless user experience.

Horizontal scalability offers several benefits:

- **Increased performance and throughput:** By adding more instances of a microservice, you can process a larger number of requests concurrently, thereby improving the overall system performance and throughput. The workload is distributed across multiple instances, reducing the processing burden on each individual microservice.
- **Improved fault tolerance and availability:** With horizontal scalability, if one instance of a microservice fails or experiences issues, the other instances can continue to handle the requests, ensuring high availability and fault tolerance. This redundancy helps prevent a single point of failure and minimizes the impact of failures on the overall system.
- **Load balancing:** Horizontal scalability facilitates load balancing as requests can be distributed across multiple instances using various load-balancing strategies. This ensures that each instance is utilized optimally and prevents overload on any specific microservice instance.
- **Elasticity:** Horizontal scalability enables elasticity, allowing you to scale the number of microservice instances up or down based on demand. You can add or remove instances dynamically to accommodate varying workloads, ensuring efficient resource utilization.

However, horizontal scalability also comes with some considerations:

- **Increased complexity:** Managing multiple instances of microservices introduces complexity in terms of deployment, configuration, and synchronization between instances. Proper orchestration and management tools are required to handle the scalability and ensure consistency across instances.
- **Communication and consistency:** When horizontally scaling microservices, you need to ensure that communication and data consistency are maintained between instances. Techniques such as messaging queues or distributed databases can be used to synchronize data across instances.
- **State management:** If a microservice maintains stateful information, such as user sessions, horizontally scaling that microservice requires careful consideration of state

management strategies. Session affinity or distributed session management techniques can be employed to handle state across instances.

Horizontal scalability is particularly effective when the workload of a microservice can be divided and processed independently by multiple instances. By distributing the workload across instances, you can achieve improved performance, fault tolerance, and flexibility in handling varying workloads in microservices-based applications.

Vertical scalability

Vertical scalability, also known as **scaling up**, *is a strategy for increasing the capacity of individual microservices by allocating more resources to them.* In the context of microservices, vertical scalability involves upgrading the hardware or adjusting the resources allocated to a specific microservice to handle increased workloads or performance requirements. You can see how vertical scalability works in the figure below:

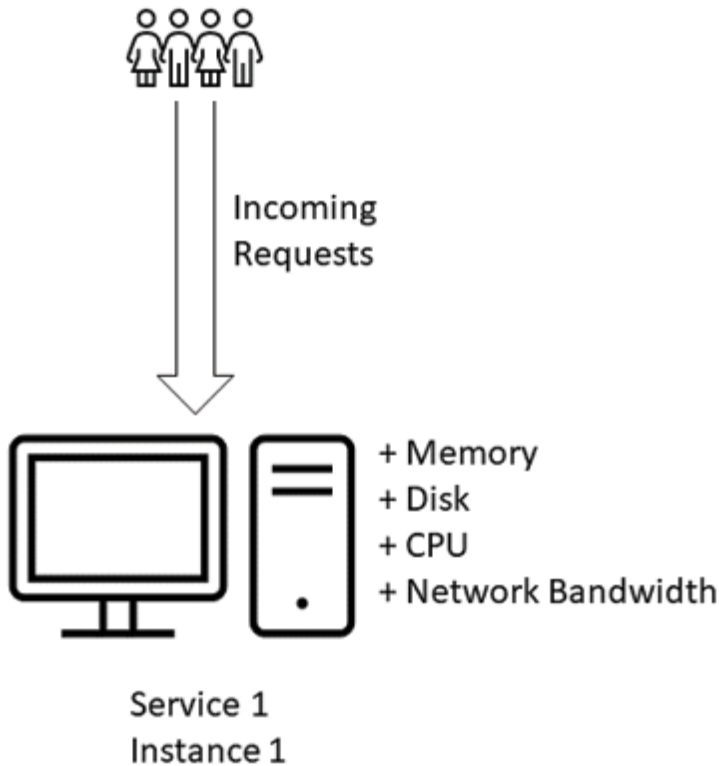


Figure 10.2: Visual explanation of how the Vertical Scalability works

When vertically scaling a microservice, you focus on enhancing its capabilities by increasing factors such as CPU power, memory, storage capacity, or network bandwidth. This can be achieved by upgrading the underlying infrastructure, such as migrating to more powerful servers, adding more RAM, or increasing the CPU cores.

The following are the benefits offered by vertical scalability:

- **Improved performance:** By allocating additional resources to a microservice, you enhance its processing power, enabling it to handle more concurrent requests and perform complex computations efficiently. This results in improved response times and overall system performance.
- **Simplified maintenance:** With vertical scalability, you deal with a single instance of the microservice, making it easier to manage and maintain. Upgrades, patches, and monitoring efforts can be focused on a single, vertically scaled

microservice instead of managing multiple instances.

- **Cost-effectiveness:** In some cases, vertical scalability can be more cost-effective than horizontal scalability, as you can leverage existing infrastructure by upgrading it rather than investing in additional servers or virtual machines.

However, vertical scalability also has its limitations:

- **Limited ceiling:** There is a maximum limit to how much a single instance of a microservice can scale vertically. Eventually, you may reach a point where further vertical scaling becomes impractical or cost-prohibitive.
- **Single point of failure:** As vertical scalability focuses on a single instance of a microservice, if that instance fails, the entire service may go down. Redundancy measures and fault-tolerant mechanisms need to be in place to mitigate this risk.
- **Limited flexibility:** Vertical scalability is applicable to individual microservices, which means each microservice may have unique scalability requirements. It may be challenging to independently scale different microservices that have varying resource needs.

Vertical scalability is most suitable when a microservice's performance is bottlenecked by its resource limitations. By upgrading the resources allocated to a microservice, you can enhance its capabilities and handle increased workloads efficiently. However, it is essential to carefully monitor resource usage and plan for any potential bottlenecks or limitations that may arise in the future.

Orchestrators

Orchestrators play a crucial role in the scalability of microservices-based applications by providing containerization and orchestration capabilities. They help manage and coordinate the deployment, scaling, and management of microservices across a cluster of machines. Some popular orchestrators include Kubernetes, Service Fabric, Docker Swarm, and Apache Mesos.

The main roles of orchestrators when talking about microservices are the ones as follows:

- **Deployment and containerization:** Orchestrators facilitate

the deployment of microservices by encapsulating them into containers. Containers provide a lightweight and isolated runtime environment for microservices, ensuring consistent deployment across different environments and simplifying the packaging and distribution process.

- **Scalability and load balancing:** Orchestrators enable horizontal scalability by automatically scaling the number of microservice instances based on demand. They monitor the resource usage and can dynamically add or remove instances to balance the workload and ensure optimal resource utilization. Load balancing techniques, such as round-robin or least-connection, are employed to distribute incoming requests across the available microservice instances.
- **Service discovery:** Orchestrators assist in the discovery and registration of microservices within the system. They provide mechanisms for microservices to discover and communicate with each other, allowing for dynamic scaling and seamless communication between services. Service registries and DNS-based service discovery are commonly used to facilitate service discovery in orchestrator environments.
- **Health monitoring and self-healing:** Orchestrators continuously monitor the health and availability of microservices. They can detect failures or unresponsive instances and automatically perform self-healing actions, such as restarting or rescheduling the failed instances on healthy nodes. This helps ensure high availability and resilience of the overall system.
- **Configuration management:** Orchestrators provide capabilities for managing the configuration of microservices across different environments. They enable centralized configuration management, allowing for dynamic updates and ensuring consistency in configurations across instances.
- **Rolling deployments and versioning:** Orchestrators support rolling deployments, allowing new versions of microservices to be deployed gradually without causing downtime or disruption to the system. They enable rolling updates, canary deployments, or blue-green deployments, ensuring smooth transitions between different versions of microservices.
- **Resource management:** Orchestrators help optimize resource allocation by managing the allocation of computing resources, such as CPU, memory, and storage, to microservices instances.

They ensure that resources are allocated efficiently based on the workload and prioritize critical services.

By leveraging orchestrators, microservices architectures can achieve enhanced scalability, flexibility, and manageability. Orchestrators simplify the management of microservices deployment, scaling, and maintenance tasks, enabling efficient utilization of resources, seamless communication between services, and robust fault tolerance mechanisms.

Most used architectural patterns

In the context of implementing microservices with Web APIs, it is essential to consider various architectural patterns that can enhance the design, scalability, and maintainability of the overall system. This section focuses on discussing some of the most used architectural patterns: **Backend for Frontend (BFF)**, **Command and Query Responsibility Segregation (CQRS)**, **Domain-Driven Design (DDD)**, and circuit breaker.

These architectural patterns provide valuable guidance and best practices for addressing specific challenges in microservices-based applications. Each pattern brings its own set of benefits and considerations, allowing developers to make informed decisions when designing and implementing their microservices architecture.

In the following sections, we will explore these patterns in detail, understanding their core concepts, benefits, and how they can be effectively applied in microservices environments. By familiarizing ourselves with these architectural patterns, we can leverage their advantages to build scalable, resilient, and maintainable microservices-based applications.

Backend for frontend

The **Backend for Frontend (BFF)** design pattern is a common architectural pattern used in microservices-based applications to improve the efficiency and flexibility of the communication between front-end clients and the microservices backend.

In a microservices architecture, different front-end applications or clients may have specific requirements or preferences when it comes to data retrieval and processing. The BFF pattern aims to

address these concerns by introducing an intermediary layer known as the **Backend for Frontend**.

The main idea behind the BFF pattern is to create a dedicated backend service for each front-end client or user interface. This backend service acts as a gateway or proxy between the client and the underlying microservices. It is responsible for aggregating, transforming, and adapting the data retrieved from multiple microservices into a format that is optimized for the specific needs of the client.

The BFF pattern offers several benefits:

- **Customized data and presentation:** Each front-end client may have unique requirements for data retrieval and presentation. The BFF service allows customization of the data and presentation logic to match the specific needs of each client. It can aggregate data from multiple microservices, perform data transformations, and format the response according to the client's requirements, reducing the amount of unnecessary data transfer and improving the performance of the client application.
- **Reduced client complexity:** By providing a dedicated backend service for each client, the BFF pattern simplifies the client-side code and reduces the complexity of handling various microservice interactions. The client application only needs to communicate with the BFF service, which abstracts the complexities of dealing with multiple microservices and their APIs.
- **Improved performance:** The BFF service can optimize data retrieval by fetching the necessary data from multiple microservices in parallel, minimizing the number of round-trips between the client and the backend. It can also implement caching mechanisms or pre-fetching strategies to further enhance performance.
- **Security and authorization:** The BFF service can centralize security and authorization logic, handling authentication and authorization for client requests. It can enforce security policies, validate user permissions, and ensure that the client only receives the data it is authorized to access.
- **Versioning and evolution:** The BFF pattern allows for independent versioning and evolution of the backend services for different front-end clients. Each BFF service can be

updated or modified to meet the evolving needs of a specific client without affecting other clients or the underlying microservices.

It is important to note that the BFF pattern introduces an additional layer of complexity and requires careful design and maintenance. However, when implemented effectively, it can greatly improve the overall performance, customization, and security of microservices-based applications by providing tailored backend services for each front-end client.

Command Query Responsibility Segregation

The **Command Query Responsibility Segregation (CQRS)** design pattern is a widely used architectural pattern in microservices-based applications that separates the concerns of read and write operations by using separate models and channels.

In traditional monolithic architectures, the same data model is often used for both read and write operations. However, as applications grow in complexity and scalability requirements increase, it can be beneficial to segregate the handling of commands (write operations) from queries (read operations) to optimize performance, scalability, and maintainability. This is where the CQRS pattern comes into play.

In the CQRS pattern, the application's data model is divided into two separate models: the command model and the query model:

- **Command model:**

- The Command Model is responsible for handling write operations or commands that modify the application's state.
- It encapsulates the business logic and validation rules necessary to process and apply the commands.
- It updates the data store and emits events or notifications to communicate changes that have occurred.
- It focuses on consistency and transactional integrity, ensuring that all changes are successfully applied.

- **Query model:**

- The Query Model is responsible for handling read operations or queries that retrieve data from the

application's state.

- It provides optimized data representations or projections specifically designed for efficient querying and retrieval.
- It can denormalize or transform data to support different read use cases and improve query performance.
- It may use specialized data stores or caching mechanisms optimized for read-intensive operations.

The separation of the command model and query model in CQRS brings several benefits:

- **Performance and scalability:** By segregating read and write operations, you can optimize the data models and storage mechanisms for their specific purposes. This allows for scaling each model independently based on the workload characteristics. The Query Model can be heavily optimized for fast and efficient reads, while the Command Model can prioritize consistency and transactional operations.
- **Enhanced flexibility:** CQRS enables the customization of data representations and projections specifically tailored to different read use cases. This allows for optimized querying, faster response times, and improved user experiences.
- **Decoupling and independence:** The separation of the command and query models allows for decoupling and independence in their development, deployment, and scalability. Each model can be developed and evolved separately, enabling flexibility and agility in implementing new features or scaling components as needed.
- **Event-driven architecture:** CQRS is often closely associated with event-driven architectures. Events are emitted by the command model when changes occur and can be consumed by other microservices or components for real-time updates, further enhancing decoupling and responsiveness.
- **Alignment with microservices:** CQRS aligns well with the principles of microservices architecture. Each microservice can have its own command and query models, enabling teams to focus on specific responsibilities and making it easier to scale and evolve individual microservices.

It is worth noting that implementing CQRS introduces additional complexity compared to a traditional CRUD-based approach. It requires careful consideration of data consistency, event-driven

communication, and synchronization between the command and query models. However, when used appropriately in complex and scalable applications, CQRS can provide significant performance and flexibility benefits, allowing for efficient read and write operations in microservices-based architectures.

Domain Driven Design

Domain Driven Design (DDD) is a design pattern and methodology that focuses on understanding and modeling the core domain of a software application. It emphasizes a collaborative approach between domain experts, developers, and stakeholders to capture the business domain and express it in the software design.

When applied to microservices architecture, DDD provides a set of principles and patterns to design individual microservices and their interactions. Here is an overview of the DDD design pattern in microservices:

- **Bounded context:**

- DDD defines the concept of a Bounded Context, which represents a distinct business domain within an application.
- Each microservice in a microservices architecture typically corresponds to a Bounded Context.
- Bounded Contexts encapsulate the domain logic and define the boundaries of consistency and language within that context.

- **Aggregate:**

- An Aggregate is a cluster of related objects within a Bounded Context that is treated as a single unit.
- Aggregates encapsulate the domain logic, maintain consistency boundaries, and ensure invariants.
- Microservices typically align with one or more Aggregates, focusing on specific business capabilities.

- **Context mapping:**

- Context mapping refers to the techniques used to define relationships and interactions between bounded contexts.
- It defines how microservices communicate and collaborate with each other.

- Different context mapping patterns, such as **Shared Kernel**, **Customer-Supplier**, or **Anti-Corruption Layer**, can be applied based on the specific integration requirements between microservices.
- **Ubiquitous language:**
 - DDD emphasizes the use of a shared language, known as the Ubiquitous language, between domain experts and developers.
 - The Ubiquitous language is used to model the domain concepts, behaviors, and relationships within the microservices.
 - It helps align the business understanding with the software implementation and fosters effective communication and collaboration.
- **Aggregate design:**
 - DDD provides guidelines for designing Aggregates, including identifying Aggregates, defining Aggregate roots, and managing consistency within Aggregates.
 - Aggregates should be designed to ensure transactional consistency and enforce business rules within the boundaries of the microservices.
- **Domain events:**
 - Domain events are an important aspect of DDD and microservices communication.
 - Domain events represent meaningful occurrences within the domain and can be used for inter-microservice communication and eventual consistency.
 - Events are published by microservices and can be consumed by other microservices to trigger actions or update their local state.
- **Strategic design:**
 - DDD encourages strategic design decisions to align the overall microservices architecture with the business goals.
 - Strategic design involves defining the context mapping patterns, Aggregate boundaries, and overall system architecture to ensure scalability, maintainability, and business agility.

By applying DDD in microservices architecture, the focus is shifted to understanding and modeling the core domain, which leads to more maintainable, scalable, and business-aligned microservices. DDD helps capture business requirements accurately, enables effective collaboration between domain experts and developers, and ensures that the microservices architecture reflects the intricacies of the business domain.

Case study

In this case study, we are implementing the **Backend for Frontend (BFF)** design pattern, where a Weather Microservice acts as the frontend service for two distinct BFFs. The BFFs cater to clients with different weather preferences, one focusing on hot weather and the other on cold weather. Additionally, we incorporate an event processor to handle asynchronous requests efficiently.

For clients living in hot weather locations, one BFF is dedicated to serving their specific needs. It orchestrates requests and retrieves relevant data from the underlying microservices, aggregating and transforming it into a format suitable for the client. Similarly, the BFF for clients favoring cold weather provides a specialized interface and retrieves data specific to their preferences.

To handle asynchronous requests and events, we employ an event processor. This component efficiently processes events in an asynchronous manner, ensuring that the BFFs can handle concurrent requests and maintain responsiveness. The event processor plays a vital role in managing the flow of data, processing events in the background while allowing the BFFs to remain performant and scalable.

Overall, the combination of the BFF design pattern, weather microservice, and event processor enables us to deliver customized weather information to clients based on their preferences. It ensures a seamless user experience by abstracting complexities, handling asynchronous requests, and providing tailored responses for hot and cold weather clients as we can see this process described better in the figure below:

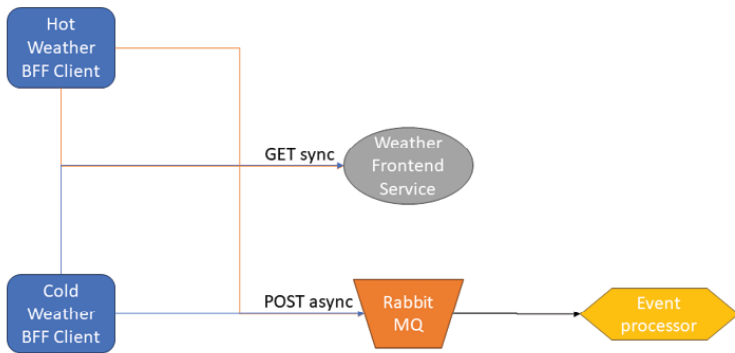


Figure 10.3: Diagram representing the practical study case.

To start our practical case study, follow the steps below:

1. Add a new Web API project to be our front-end microservice.
2. Add new Console Application Project, in this new console application project we should add the following packages:
 - Add nuget package
 - **RabbitMQ.Client**
5. Add two new Web API projects, they are going to be our BFFs. We have named it as BFFOne and BFFTwo:
 - Add nuget packages
 - **Newtonsoft.Json**
 - **RabbitMQ.Client**

In the end, this is the project solution that we have. The 3 Web APIs are basically the same with a console application as we can see in the figure below:

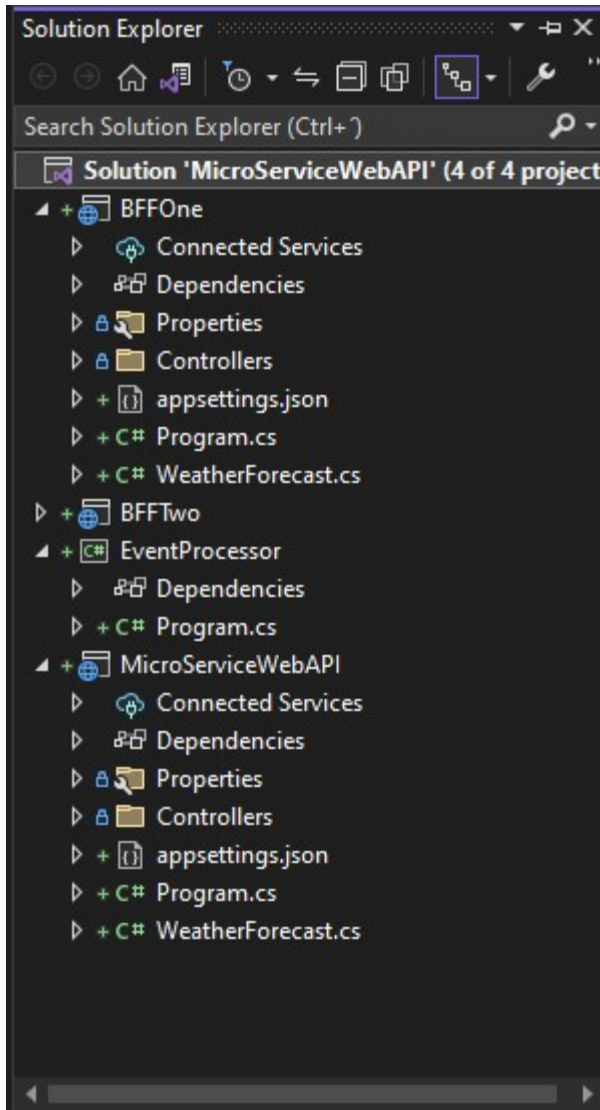


Figure 10.4: Solution explorer with all the 4 projects.

Now, let us modify our projects to apply the BFF design pattern with asynchronous communication. A few modifications are needed to represent the design pattern.

First, let us adjust the **MicroServiceWebAPI** project. We have broken the weather summaries into hot and cold summaries, and those summaries will work as our database.

This is the **WeatherForecastController** for the **MicroServiceWebAPI** project.

```
1. [ApiController]
2.     [Route("[controller]")]
3.     public class WeatherForecastController
4.     : ControllerBase
5.     {
6.         private static readonly string[]
7. HotSummaries = new[]
8.         {
9.             "Cool", "Mild", "Warm",
10.            "Balmy", "Hot",
11.            "Sweltering", "Scorching" };
12.
13.         private static readonly string[]
14. ColdSummaries = new[]
15.         {
16.             "Freezing", "Bracing",
17.            "Chilly", "Cool", "Mild",
18.            "Warm", "Balmy" };
19.
20.         private readonly
21. ILogger<WeatherForecastController> _logger;
22.
23.         public WeatherForecastController(
24.             ILogger<WeatherForecastController> logger)
25.         {
26.             _logger = logger;
27.         }
28.
29.         [HttpGet(Name =
30. "GetWeatherForecast")]
31.         public IEnumerable<WeatherForecast>
32. Get([FromQuery]
33.         string weather)
34.         {
35.             switch (weather.ToLower())
```

```

26.         {
27.             case "cold":
28.                 return
29.                 Enumerable.Range(1, 5).Select(index =>
30.                     new WeatherForecast
31.                     {
32.                         Date =
33.                         DateTime.Now.AddDays(index),
34.                         TemperatureC =
35.                         Random.Shared.Next(-20, 15),
36.                         Summary =
37.                         ColdSummaries[Random.Shared
38.                             .Next(ColdSummaries.Length)]
39.                     }).ToArray();
40.             case "hot":
41.                 return
42.                 Enumerable.Range(1, 5).Select(index =>
43.                     new WeatherForecast
44.                     {
45.                         Date =
46.                         DateTime.Now.AddDays(index),
47.                         TemperatureC =
48.                         Random.Shared.Next(15, 55),
49.                         Summary =
50.                         HotSummaries[Random.Shared
51.                             .Next(HotSummaries.Length)]
52.                     }).ToArray(); ;
53.             default:
54.                 return
55.                 Enumerable.Empty<WeatherForecast>();
56.         }
57.     }
58. }
59. }

```

A small change was also made in the **WeatherForecast** class for all the 3 projects:

```
1. public class WeatherForecast
2. {
3.     public DateTime Date { get; set; }
4.
5.     public int TemperatureC { get; set; }
6.
7.     public int TemperatureF => 32 +
   (int) (TemperatureC / 0.5556);
8.
9.     public string? Summary { get; set; }
10. }
```

Now, let us update both BFFs controllers. Each BFF sends specific information about his weather to the **EventProcessor** when posting Weather and to the client microservice when getting weather.

The HTTP GET adds its specific information and makes a HTTP request to the Weather Microservice.

The HTTP POST adds its specific information and publishes a message in the RabbitMQ queue, this message will be handled by the **EventProcessor**.

The **WeatherForecastController** from the Hot Weather BFF, specifically designed for handling hot weather scenarios, as you can see in the code below:

```
1. [ApiController]
2. [Route("[controller]")]
3. public class WeatherForecastController
   : ControllerBase
4. {
5.
6.     private readonly
   ILogger<WeatherForecastController> _logger;
7. }
```



```

8.         public WeatherForecastController(
9.             ILogger<WeatherForecastController> logger)
10.        {
11.            _logger = logger;
12.        }
13.
14.        [HttpGet (Name =
15.            "GetWeatherForecast")]
16.        public async
17.            Task<IEnumerable<WeatherForecast>> Get ()
18.        {
19.            var result = new
20.                List<WeatherForecast>();
21.            string baseUrl = "https://
22.                localhost:7173/";
23.            string url = baseUrl +
24.                "WeatherForecast?weather=hot";
25.            using (HttpClient client = new
26.                HttpClient())
27.            {
28.                using (HttpResponseMessage
29.                    responseMessage =
30.                        await
31.                            client.GetAsync(url))
32.                {
33.                    using (HttpContent
34.                        content = responseMessage
35.                            .Content)
36.                    {
37.                        string data = await
38.                            content
39.                                .ReadAsStringAsync();
40.                        result =
41.                            JsonConvert

```

```

31.         .DeserializeObject<List<WeatherForecast>>(data);
32.     }
33. }
34. }
35.     return result;
36. }
37. [HttpPost]
38.     public void Post([FromBody]
WeatherForecast weatherForecast)
39.     {
40.         var factory = new
ConnectionFactory() { HostName
41.             = "localhost" };
42.         using (var connection =
factory.CreateConnection())
43.         using (var channel =
connection.CreateModel())
44.         {
45.             channel.QueueDeclare(
46.                 queue: "weatherForecastSampleQueue",
47.                 durable: false, exclusive:
false, autoDelete: false,
48.                 arguments: null);
49.
50.             string message = "From BFF
One, Date: "
51.                 + weatherForecast.Date + ",
Temperature in C°: "
52.                 +
weatherForecast.TemperatureC + " and
Summary: "
53.                 + weatherForecast.Summary;
54.
55.             var body =
Encoding.UTF8.GetBytes(message);

```

```

56.
57. channel.BasicPublish(exchange: "",
58.                       routingKey:
59. "weatherForecastSampleQueue",
60.                       basicProperties:
61. null, body: body);
62.     }
63. }
64. }

```

The **WeatherForecastController** from the Cold Weather BFF, specifically designed for handling cold weather scenarios:

```

1. [ApiController]
2. [Route("[controller]")]
3. public class WeatherForecastController
4. : ControllerBase
5. {
6.     private readonly
7. ILogger<WeatherForecastController> _logger;
8.     public WeatherForecastController(
9. ILogger<WeatherForecastController> logger)
10.    {
11.        _logger = logger;
12.    }
13.
14.    [HttpGet(Name =
15. "GetWeatherForecast")]
16.    public async Task<
17. IEnumerable<WeatherForecast>> Get()
18.    {
19.        var result = new
20. List<WeatherForecast>();
21.        string baseUrl = "https://"

```

```

localhost:7173/";
19.         string url = baseUrl +
    "WeatherForecast?weather=cold";
20.         using (HttpClient client = new
    HttpClient())
21.         {
22.             using (HttpResponseMessage
    responseMessage =
23.                 await
    client.GetAsync(url))
24.             {
25.                 using (HttpContent
    content = responseMessage
26.                     .Content)
27.                 {
28.                     string data = await
    content
29.                         .ReadAsStringAsync();
30.                     result =
    JsonConvert
31.                         .DeserializeObject<List<WeatherForecast>>(data);
32.                 }
33.             }
34.         }
35.         return result;
36.     }
37.     [HttpPost]
38.     public void Post([FromBody]
    WeatherForecast weatherForecast)
39.     {
40.         var factory = new
    ConnectionFactory()
41.             { HostName =
    "localhost" };
42.         using (var connection =

```

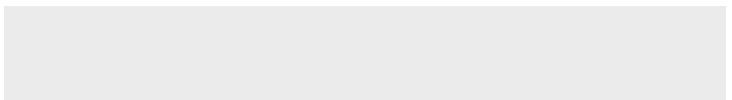
```

factory.CreateConnection())
43.         using (var channel =
connection.CreateModel())
44.         {
45.             channel.QueueDeclare(
46.                 queue:
"weatherForecastSampleQueue",
47.                 durable: false, exclusive:
false, autoDelete: false,
48.                 arguments: null);
49.
50.                 string message = "From BFF
Two, Date: "
51.                 + weatherForecast.Date +
", Temperature in C°: "
52.                 +
weatherForecast.TemperatureC + " and
Summary: "
53.                 +
weatherForecast.Summary;
54.
55.                 var body =
Encoding.UTF8.GetBytes(message);
56.
57.                 channel.BasicPublish(exchange: "",
58.                                     routingKey:
"weatherForecastSampleQueue",
59.                                     basicProperties:
null, body: body);
60.             }
61.         }
62.     }

```

The event processor is responsible for subscribing to a queue and processing its incoming messages.

This is the **Program.cs** from the event processor:



```
1. // See https://aka.ms/new-console-template for more
   information
2. using RabbitMQ.Client;
3. using RabbitMQ.Client.Events;
4. using System;
5. using System.Text;
6. using System.Threading;
7. using System.Threading.Tasks;
8.
9. Console.WriteLine("Hello, World!");
10. var factory = new ConnectionFactory() {
    HostName = "localhost" };
11. string queueName =
    "weatherForecastSampleQueue";
12. using (var rabbitMqConnection =
    factory.CreateConnection())
13. {
14.
15.     using (var rabbitMqChannel =
    rabbitMqConnection.CreateModel())
16.     {
17.         Thread.Sleep(5000);
18.
19.         rabbitMqChannel.QueueDeclare(queue:
    queueName,
20.                                     durable:
    false,
21.                                     exclusive:
    false,
22.                                     autoDelete:
    false,
23.                                     arguments:
    null);
24.
25.         rabbitMqChannel.BasicQos(prefetchSize: 0,
    prefetchCount: 1,
```

```

26.         global: false);
27.
28.         int messageCount =
Convert.ToInt16(rabbitMqChannel
29.             .MessageCount(queueName));
30.         Console.WriteLine(" Listening to
the queue.
31.         This channels has {0} messages on
the queue», messageCount);
32.
33.         var consumer = new
EventingBasicConsumer(rabbitMqChannel);
34.         consumer.Received += (model,
ea) =>
35.         {
36.             var message =
Encoding.UTF8.GetString(ea.Body
37.                 .ToArray());
38.             Console.WriteLine(" Weather
Forecast received: "
39.                 + message);
40.
41.             rabbitMqChannel.BasicAck(deliveryTag:
ea.DeliveryTag,
42.                 multiple:
false);
43.             Thread.Sleep(1000);
44.         };
45.         rabbitMqChannel.BasicConsume(queue:
queueName,
46.             autoAck:
false,
47.             consumer:
consumer);
48.     }

```

Before pushing *F5* and running the projects, you must configure the solution to debug all the projects at the same time.

Right-click on the **Project solution** | **Properties**:

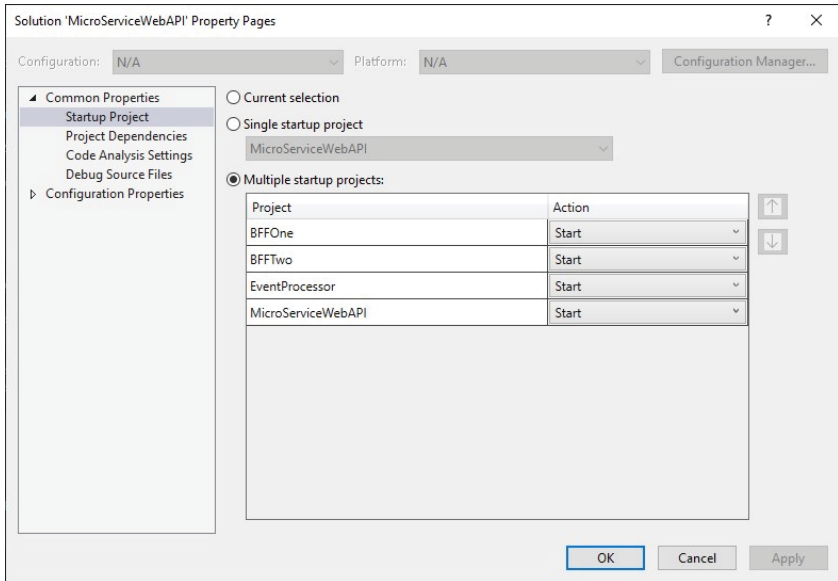


Figure 10.5: Solution properties window

From BFF 1, the hot BFF making a GET request we have the following outputs as show in the figure below from our swagger:

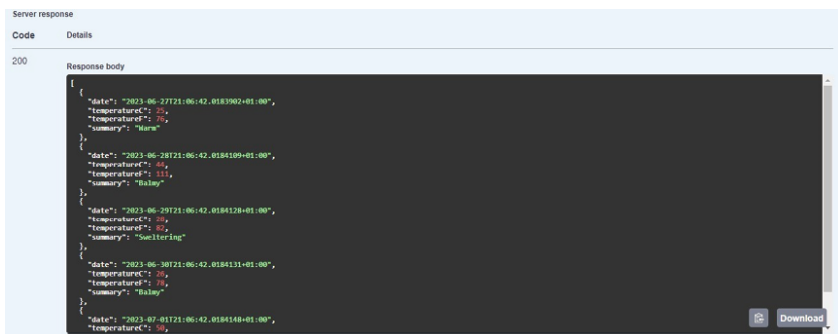


Figure 10.6: Swagger response from a GET Request for the BFF 1

BFF 1, the hot BFF making a POST request, as we can see in our

swagger from the figure below:

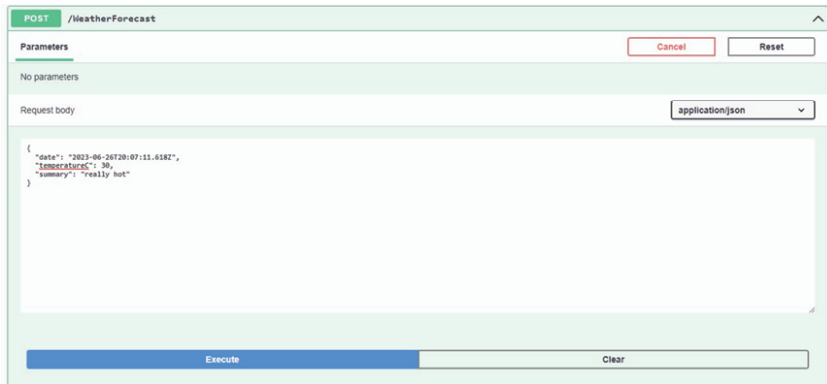


Figure 10.7: Swagger request from a POST Request for the BFF 1

Following is the answer from the event processor:

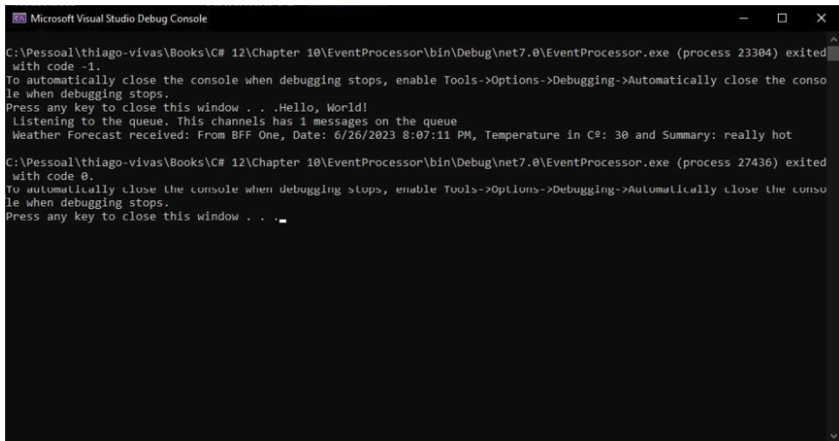


Figure 10.8: Console application displaying the processed message from BFF 1 POST operation

From our BFF 2, the cold BFF. We are making a GET request from our swagger as you can see the response from the figure below:

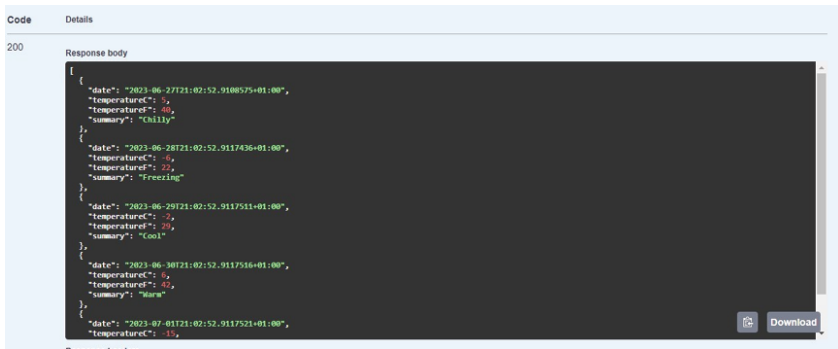


Figure 10.9: Swagger response from a GET Request for the BFF 2

In the BFF 2, the cold weather BFF, we are making a POST request from our swagger as you can see from the figure below:

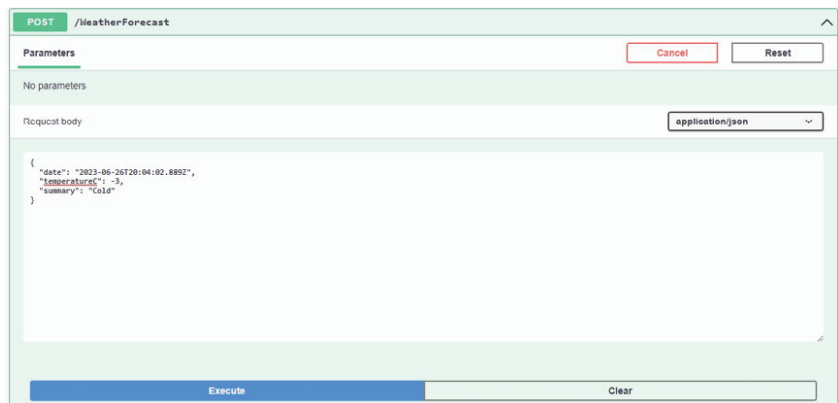
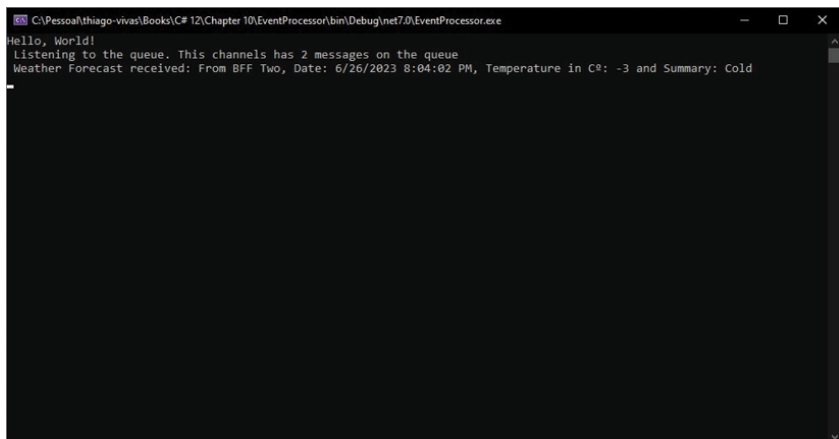


Figure 10.10: Swagger request from a POST Request for the BFF 2

We will get the following answer from the event processor:



```
C:\Pessoal\thiago-vivas\Books\C# 12\Chapter 10\EventProcessor\bin\Debug\net7.0\EventProcessor.exe
Hello, World!
Listening to the queue. This channels has 2 messages on the queue
Weather Forecast received: From BFF Two, Date: 6/26/2023 8:04:02 PM, Temperature in Cº: -3 and Summary: Cold
```

Figure 10.11: Console application displaying the processed message from BFF 2 POST operation

Conclusion

In conclusion, this chapter has provided an in-depth guide to building scalable and resilient microservices using Web APIs. We began by exploring the benefits of microservices architecture and the importance of Web APIs in facilitating communication between microservices. Throughout the chapter, we focused on the critical challenge of scaling microservices-based applications.

We discussed the various scaling techniques available, including horizontal scaling, vertical scaling, and auto-scaling. Step-by-step instructions were provided for implementing each technique, along with best practices and common pitfalls to avoid. We also delved into architectural patterns commonly used in microservices and their role in scalability and resilience.

The practical case study presented a real-world scenario for building a microservices-based application. By following the case study, readers gained hands-on experience in designing a scalable microservices architecture, implementing horizontal scaling, and utilizing Web APIs for inter-microservice communication.

Additionally, we explored communication patterns between microservices, including synchronous and asynchronous approaches. We discussed the importance of choosing the appropriate communication mechanism based on the specific

requirements of the application.

By mastering the concepts and techniques presented in this chapter, readers are now equipped with the knowledge and skills necessary to tackle the challenges of building scalable microservices architectures. They have gained a solid understanding of scaling techniques, architectural patterns, and effective communication strategies, enabling them to build resilient and highly scalable microservices-based applications.

As microservices continue to gain popularity in the software development landscape, the ability to design, implement, and scale microservices effectively becomes increasingly crucial. The knowledge gained from this chapter will empower readers to create robust microservices architectures that can handle growing workloads and adapt to changing demands.

In conclusion, building scalable and resilient microservices using Web APIs requires careful consideration of architectural design, scaling techniques, and communication patterns. Armed with the insights and practical guidance provided in this chapter, readers are well-prepared to embark on their journey of implementing microservices-based applications with confidence and success.

In the upcoming chapter, we venture into the dynamic landscape of **Continuous Integration and Continuous Deployment (CI/CD)** with Docker and Azure DevOps. CI/CD form the backbone of modern software development, and this chapter explores their seamless integration with Docker technology and Azure DevOps services. We delve into the pivotal role Docker plays in containerization, ensuring consistency across diverse environments. The synergy with Azure DevOps amplifies the efficiency of the CI/CD pipeline, enabling automated testing, deployment, and delivery. Join us as we unravel the power of this integration, providing practical insights and strategies to streamline your development workflows and enhance the agility of your software delivery process.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 11

CI/CD with Docker and Azure DevOps

Introduction

This chapter provides a practical guide to implementing a **continuous integration and continuous deployment (CI/CD)** pipeline for containerized applications using Docker and Azure DevOps. The chapter begins by providing an overview of Docker and its role in containerization. It then introduces Azure DevOps and explains how it can be used to automate the CI/CD process for containerized applications.

The chapter then walks through the various Docker commands required for building and deploying containerized applications. It explains how to build Docker images, push them to Docker Hub, and deploy them to an Azure container registry. The chapter then provides an overview of continuous integration and continuous deployment and how it can be used to streamline the application development and deployment process. It explains how Azure DevOps can be used to automate the CI/CD process, including topics like configuring build pipelines, release pipelines, and environment management.

To provide a practical case study, the chapter walks through building a sample containerized application, setting up a CI/CD pipeline using Azure DevOps, and deploying the application to a production environment.

Structure

This chapter covers the following topics:

- Overview
- Docker
 - Docker containers
 - Container images
 - Docker images
 - Container images X Docker images
 - Docker advantages for your apps
 - Understanding the Dockerfile
 - Docker commands
- Azure DevOps
- Continuous integration
 - Benefits of continuous integration
- Continuous deployment
 - Benefits of continuous deployment
- Case study
 - Creating a Dockerfile
 - Creating a Docker image
 - Applying continuous integration
 - Applying continuous deployment

Objectives

In this chapter, readers will grasp the fundamentals of Docker's role in containerization and delve into Azure DevOps as a comprehensive DevOps platform. Exploring Docker commands, they will learn to build and deploy containerized applications,

understanding CI significance in software development. Configuring Azure DevOps pipelines automates CI processes, leading to insights into CD benefits. The chapter showcases Azure DevOps' role in CD pipeline automation, emphasizing release pipelines and environment management. Through hands-on experience and a practical case study, readers will acquire skills to set up CI/CD pipelines efficiently using Docker and Azure DevOps, ensuring a comprehensive understanding of containerization and DevOps practices.

Overview

CI and CD with Docker in Azure DevOps enable teams to automate the build, test, and deployment of containerized applications. It leverages the capabilities of Docker, a popular containerization platform, and Azure DevOps, a comprehensive DevOps toolset provided by Microsoft.

By integrating Docker into Azure DevOps pipelines, developers can easily build Docker images, push them to Docker registries, and deploy them to various environments, including development, staging, and production. This integration facilitates consistent and reliable application deployments in a containerized environment.

With CI, code changes are automatically built, tested, and integrated into a shared repository whenever developers push their code. This ensures that the application remains in a continuously deployable state, enabling faster feedback and reduced integration issues.

CD takes CI a step further by automating the deployment process, allowing for seamless delivery of containerized applications to different environments. Azure DevOps provides features for configuring release pipelines, defining deployment strategies, and orchestrating the deployment of Docker containers to **Azure Kubernetes Service (AKS)**, Azure App Service, or other target platforms.

The combination of Docker and Azure DevOps also offers benefits such as scalability, portability, and reproducibility. Docker's containerization technology enables applications to run consistently across different environments, reducing deployment and

compatibility challenges. Azure DevOps provides a centralized platform for managing CI and CD pipelines, fostering collaboration, version control, and monitoring the entire application lifecycle.

By utilizing CI and CD with Docker in Azure DevOps, development teams can achieve faster time-to-market, improved software quality, and greater agility in responding to customer needs. It empowers teams to focus on delivering value to users while ensuring consistency, reliability, and efficiency in the deployment process.

Docker

Docker is an open-source platform that revolutionizes the development, packaging, and execution of applications. Its swift setup process allows for easy creation and deployment of Docker containers, providing a time advantage in managing environments efficiently.

One of the key advantages of Docker is its ability to package applications into Docker images, which can then be run within Docker containers. These containers operate in isolation from one another, ensuring that they can have distinct or identical configurations while maintaining consistent behavior across all instances of the Docker images.

By utilizing Docker, developers can encapsulate their applications into portable and self-contained units, making it effortless to reproduce and deploy them across different environments. This eliminates compatibility issues and guarantees consistency in the behavior of the applications, regardless of the underlying infrastructure.

Furthermore, Docker simplifies the management of dependencies and ensures the reproducibility of the application's runtime environment. Developers can specify the necessary dependencies within the Docker image, making it easier to maintain and distribute the application across various platforms.

In summary, Docker provides developers with a powerful toolset for packaging and deploying applications, enhancing efficiency, portability, and reproducibility. By leveraging Docker's containerization technology, developers can streamline their development processes and confidently deploy applications with

consistent behavior across diverse environments.

Docker containers

A **Docker container** functions as a self-contained entity, with its processes separate from others running on the host machine. Each container operates within its own environment, allowing for individualized settings without impacting or being affected by external processes.

Key functionalities of Docker containers include:

- **Cross-platform compatibility:** Docker containers can run on any operating system, providing flexibility and portability across different environments.
- **Isolation and control:** Each container operates within its isolated and independent settings, creating a controlled environment where applications can run reliably without interference from external factors.
- **Versatility:** Docker containers can be deployed on various machines, including virtual machines or cloud instances, enabling seamless deployment across different infrastructures.
- **Image-based architecture:** Multiple Docker images can run concurrently within a container, facilitating the execution of diverse applications while maintaining separation and encapsulation.
- **Ease of management:** Docker containers are straightforward to handle. With a few simple commands, you can effortlessly create, start, stop, delete, or move containers, allowing for efficient management of application deployment and resource utilization.

By utilizing Docker containers, developers can leverage the advantages of isolated environments, simplifying application deployment, enhancing scalability, and ensuring consistent behavior across different systems. This approach streamlines development processes, facilitates collaboration, and promotes reproducibility, making Docker an invaluable tool in modern software development and deployment workflows.

Container images

A Docker container image has the software binaries, scripts,

configurations, and its dependencies for running equally every time it is instantiated in a Docker Container. With your Docker container image, you may easily replicate your software as many times as needed, being useful when scaling vertically or horizontally your software.

Key features and benefits of Docker container images include:

- **Configuration management:** Docker container images allow you to define and store all the required configurations for your containers. This includes environment variables, network settings, file system mappings, and more, ensuring consistency across container instances.
- **Dependency management:** By bundling dependencies within a Docker container image, you eliminate the need for manual installation and ensure that all required components are readily available. Making it easier to install your applications, with less compatibility problems.
- **Portability and reproducibility:** you can easily store and distribute your Docker container image, being able to share it in public or private repositories. Every image deployed created from this Docker container image must have the same behavior, no matter the differences across different environments.
- **Versioning and rollbacks:** Docker container images are versioned, like NuGet packages. This allows you to manage your software versions installed enabling fast rollbacks, also helping when back-tracing issues in different environments, and to organize different functionalities per version.
- **Scalability and performance:** Docker container images provide a lightweight and efficient runtime environment. As containerized applications shares the same machine hardware, meaning that the we have multiple containers running on the same host, it is easier to manage resource utilization and to scale those Docker container images.

By leveraging Docker container images, developers can streamline the deployment process, simplify application management, and promote consistency and reproducibility across different environments. With the ability to create, share, and replicate container images, Docker facilitates collaboration and accelerates the software development lifecycle.

Docker images

A Docker image encompasses all the necessary components to package and deploy your application seamlessly. It encapsulates various tasks, including restoring NuGet packages, downloading additional Docker images, building, testing, and packaging your application.

Key features and benefits of Docker images include:

- **Dependency management:** Docker images allow you to define and manage application dependencies, ensuring that all required packages and libraries are readily available within the image. This eliminates manual installation and streamlines the deployment process.
- **Reproducibility:** Docker images provide a consistent and reproducible environment for running your application. By bundling all the required tasks and dependencies within the image, you can ensure that the application behaves consistently across different environments and deployments.
- **Portability:** Docker images are portable and platform-agnostic. Once an image is created, it can be shared and deployed on any machine or platform that supports Docker, making it easy to move and run applications across various environments without compatibility issues.
- **Build and test automation:** Docker images enable the automation of build and test processes. You can define the necessary steps and commands within the image to perform tasks such as building, testing, and packaging your application, ensuring consistency and reliability throughout the development and deployment lifecycle.
- **Versioning and Rollbacks:** Docker images support versioning, allowing you to track and manage changes to the image over time. This enables seamless rollbacks to previous versions if needed, providing a safety net for managing application updates and maintaining stability.

By utilizing Docker images, developers can streamline the packaging, deployment, and testing of their applications. The encapsulation of tasks and dependencies within the image simplifies the development workflow, promotes reproducibility, and ensures consistent behavior across different environments. With Docker images, you have a powerful tool to achieve efficient and reliable

application delivery in containerized environments.

Container images X Docker images

In the context of Docker, the terms **Docker Container images** and **Docker images** are often used interchangeably. However, if we want to make a distinction, we can consider the following:

- **Docker images:** The term Docker images refers to the standardized, read-only templates that are used to create Docker containers. A Docker image is a standalone, executable package that includes everything needed to run an application, including the code, runtime, system tools, libraries, and dependencies. Docker images are created using a Dockerfile, which contains instructions on how to build the image.
- **Docker container images:** The term Docker container images can be used to specifically emphasize that we are referring to images that are used for configuring and running Docker containers. Docker container images are essentially the same as Docker images. They are portable, self-contained units that contain all the necessary components to run an application within a containerized environment.

To summarize, both Docker container images and Docker images refer to the same concept: the self-contained templates used to create Docker containers. The term Docker images is more commonly used, while Docker container images can be used to emphasize the context of using images for configuring and running containers.

Docker advantages for your apps

Containerizing your application offers numerous advantages, with the most significant ones being:

- **Fast deployment:** Containerization enables rapid deployment of your application. With Docker images, you can spin up containers within seconds, allowing for quick and efficient scaling of your application.
- **Consistent software:** Containerized applications exhibit consistent behavior across different environments. Once we have a Docker image you can be sure that this software will have the same behavior every time that it is deployed.

- **Platform-independent:** you can deploy your Docker image in any platform that supports Docker, no matter the operational system or it is deployed in your local server or in the cloud.
- **Easy vertical scaling:** Containerization simplifies vertical scaling, enabling you to scale your application up and down in near real-time. You can optimize resource utilization adapting to the workload demand through scaling down or up by adjusting the allocated resources to the containers.
- **Easy horizontal scaling:** you can easily add or remove containers with your Docker Images, maximizing resources utilization within your host, and improving performance through sharing the workload among all the containers.

By using containerization, you can take advantage of more control over your software, making it easier to manage, deploy and scale it. These advantages empower you to deliver applications quickly, maintain consistency across different environments, and optimize resource utilization for enhanced performance and scalability.

Understanding the Dockerfile

The Dockerfile plays a crucial role in the creation of Docker images for your applications. It serves as a blueprint that outlines a series of ordered steps to be executed by Docker in order to generate the desired Docker image.

By utilizing the Dockerfile, you can define the specific configuration and dependencies required for your application's image creation. Each step within the Dockerfile is carefully crafted to ensure that the image is built accurately and consistently.

The Dockerfile acts as a set of instructions, guiding Docker through the process of assembling the image layer by layer. These instructions encompass various tasks, including setting up the base image, installing necessary dependencies, copying application code, configuring runtime environments, and defining executable commands.

With the Dockerfile, you have complete control over the image creation process, allowing for customization and fine-tuning according to your application's requirements. By specifying the steps in the correct order, you can ensure that the Docker image is created with precision and efficiency.

In summary, the Dockerfile is a fundamental component in Docker's image creation workflow. It empowers you to define the necessary steps and configurations, providing a consistent and reproducible method to generate Docker images tailored specifically for your applications.

Example of a Dockerfile:

```
1. # Set the base image to use
2. FROM node:14-alpine
3.
4. # Set the working directory inside the container
5. WORKDIR /app
6.
7. # Copy the package.json and package-lock.json files
8. COPY package*.json ./
9.
10. # Install the dependencies
11. RUN npm install
12.
13. # Copy the application code into the container
14. COPY . .
15.
16. # Set the environment variable for the application
17. ENV PORT=3000
18.
19. # Expose the port the application will listen on
20. EXPOSE $PORT
21.
22. # Specify the command to run the application
23. CMD [ "node", "app.js" ]
```

In this example, the Dockerfile starts with a base image of node:14-alpine, which provides a lightweight Linux distribution with Node.js installed. The working directory is set to `/app`, `package.json`, and `package-lock.json` files are copied to the container.

Next, the dependencies are installed using the `RUN npm install` command. The application code is then copied into the container

using the **COPY** . . command.

An environment variable **PORT** is set to 3000, specifying the port on which the application will listen. The **EXPOSE** instruction exposes that port to allow external access.

Finally, the **CMD** command specifies the command to run the application, in this case `node app.js`.

This Dockerfile can be used to build a Docker image for your Node.js web application, allowing you to containerize and deploy it easily across different environments.

Dockerfile for multi-stage builds

A multi-stage Dockerfile introduces a powerful concept that enables efficient and streamlined image building. It is distinguished by the presence of multiple **FROM** statements in the Dockerfile, where each **FROM** statement represents a distinct stage of the build process. This approach allows you to leverage and reuse artifacts from previous stages, optimizing the final image's size and performance.

By employing a multi-stage build, you can take advantage of the clean separation between build-time and runtime dependencies. This separation ensures that only the necessary components are included in the final image, resulting in smaller and more secure deployments.

Here is an example of a multi-stage Dockerfile:

```
1. # Build stage
2. FROM node:14-alpine as build
3.
4. WORKDIR /app
5.
6. COPY package*.json ./
7.
8. RUN npm install
9.
10. COPY . .
11.
12. RUN npm run build
13.
```



```
14. # Production stage
15. FROM nginx:alpine as production
16.
17. COPY --from=build /app/dist /usr/share/
    nginx/html
18.
19. EXPOSE 80
20.
21. CMD ["nginx", "-g", "daemon off;"]
```

In this example, the Dockerfile consists of two stages: the build and production stages.

The build stage begins with the `node:14-alpine` base image, where the necessary dependencies are installed, the application code is copied, and the build command (`npm run build`) is executed.

The production stage starts with the `nginx:alpine` base image. It then uses the `COPY --from=build` command to copy the build artifacts from the previous stage (`/app/dist`) into the final image. This ensures that only the compiled and optimized assets are included in the production image.

The resulting image is significantly smaller and contains only the runtime dependencies required for serving the application. It can be deployed with ease and efficiency, resulting in improved performance and reduced attack surface.

Multi-stage Dockerfiles are a valuable technique for optimizing image size, enhancing security, and simplifying the deployment process. By leveraging the power of multiple stages, you can achieve more efficient and manageable Docker image builds for your applications.

Docker commands

Docker provides a comprehensive set of commands that are fundamental for creating, managing, and interacting with containers and images. These essential Docker commands empower you to effectively manage your containerized environments:

- **Docker build:** This command builds a Docker image based on the instructions specified in the Dockerfile. It is used to create

a customized image that includes all the necessary dependencies and configurations for your application.

- **Docker run:** With this command, you can instantiate a container from a specific Docker image and start running it. It sets up the container's runtime environment, network, and other configurations defined in the image, allowing your application to run in an isolated and portable manner.
- **Docker ps:** This command lists the running containers on your system. It provides valuable information such as container IDs, names, status, and resource usage. The **docker ps** command allows you to monitor and manage your running containers effectively.
- **Docker stop:** When you need to stop one or more running containers, the **docker stop** command comes in handy. It gracefully stops the specified container(s), allowing for a controlled shutdown and release of resources.
- **Docker rm:** This command enables you to remove one or more containers from your system. It permanently deletes the specified container(s) and frees up system resources. Properly cleaning up containers that are no longer needed is important for efficient resource utilization.
- **Docker rmi:** When you want to remove one or more Docker images from your local repository, the **docker rmi** command is used. It deletes the specified image(s) and frees up disk space. This command helps manage your image repository and ensures that you only retain necessary images.
- **Docker image:** The **docker image** command is a versatile tool for managing Docker images. It allows you to list, build, tag, inspect, and perform various operations related to Docker images. This command provides a range of functionalities to effectively manage your image collection.

By leveraging these key Docker commands, such as **build**, **run**, **ps**, **stop**, **rm**, **rmi**, and **image**, you can efficiently create, manage, and interact with containers and images, enabling seamless development, deployment, and maintenance of your applications.

Azure DevOps

Azure DevOps is a comprehensive set of development tools and services provided by Microsoft to support the entire DevOps

lifecycle. It enables teams to plan, develop, test, and deploy applications efficiently, fostering collaboration, automation, and continuous integration and delivery.

Azure DevOps offers a range of features and capabilities that can be used individually or as an integrated suite. These include:

- **Project management:** Azure Boards provides agile planning and tracking capabilities, allowing teams to manage work items, track progress, and visualize workflows.
- **Source control:** Azure Repos offers version control for code repositories, supporting both Git and **Team Foundation Version Control (TFVC)**. It enables collaboration, branching, and merging of code changes.
- **CI/CD:** Azure Pipelines automates the build, test, and deployment processes, allowing teams to define pipelines as code and achieve continuous integration and delivery. It supports various programming languages and platforms.
- **Artifact management:** Azure Artifacts provides a centralized repository for managing and sharing dependencies, such as NuGet packages, npm packages, and Maven artifacts. It enables easy versioning, publishing, and consumption of artifacts.
- **Test management:** Azure Test Plans facilitates test case management, exploratory testing, and test execution. It helps teams plan, track, and analyze their testing efforts, ensuring quality throughout the development lifecycle.
- **Collaboration:** Azure Boards, Azure Repos, and Azure Pipelines integrate with popular collaboration tools like Microsoft Teams, enabling seamless communication, visibility, and transparency across teams.
- **Analytics and insights:** Azure DevOps provides built-in analytics and reporting capabilities, offering insights into code quality, pipeline performance, work item tracking, and more. This data-driven approach helps teams make informed decisions and continuously improve their processes.

Azure DevOps supports various development scenarios and can be used for projects of any size, from small teams to enterprise-scale deployments. It integrates well with other Azure services and popular development tools, providing flexibility and extensibility.

By leveraging Azure DevOps, development teams can streamline their workflows, improve collaboration, automate processes, and achieve faster and more reliable delivery of software applications.

Continuous integration

Continuous integration is a crucial development practice that promotes frequent integration of code changes into a shared repository, triggered automatically whenever a developer pushes a commit. It facilitates an automated build process and allows for the use of tools that analyze code, promptly detecting and highlighting any issues that arise.

Benefits of continuous integration

We have listed below the main benefits of the continuous integration process:

- **Quicker integrations:** By integrating code changes regularly, CI minimizes the risk of conflicts and integration challenges that often arise when multiple developers are working on the same project. It ensures that changes are continuously merged and tested, enabling a smoother and more efficient development process.
- **Solid repository:** CI promotes a robust and up-to-date code repository by enforcing the practice of committing changes regularly. This helps maintain a reliable and accessible codebase, allowing teams to collaborate effectively and reduce the chances of code divergence.
- **Business rule validation with unit tests:** CI encourages the use of unit tests to validate business rules and functional requirements. By integrating tests into the CI process, developers can ensure that code changes align with expected behavior and identify potential regressions early on, promoting code reliability and reducing the time spent on debugging.
- **Early problem detection:** CI allows problems in the codebase to be identified as soon as they emerge, preventing them from snowballing into larger issues. By leveraging automated tools for code analysis, potential bugs, syntax errors, and style violations can be detected early, enabling developers to address them promptly.
- **Increased project status visibility:** CI provides enhanced

visibility into the project's status by generating build reports and notifications. This enables stakeholders to monitor the progress, stability, and quality of the application throughout the development lifecycle, fostering transparency and effective decision-making.

- **Code quality checking:** CI allows teams to incorporate code quality checking tools into the build process. These tools assess code against predefined coding standards, best practices, and performance metrics. By integrating code quality checks, developers can maintain consistent coding standards and improve the overall quality of the software.

By adopting continuous integration, development teams can experience smoother collaboration, faster integration cycles, early bug detection, improved code quality, and enhanced project transparency. These benefits ultimately lead to increased productivity, reduced development risks, and the ability to deliver reliable software solutions more efficiently.

Continuous deployment

Continuous deployment is a software delivery approach that automates the build and deployment process, enabling rapid and reliable deployment of software into production. By automating the entire deployment phase, it eliminates the need for manual and time-consuming steps, streamlining the release process and saving valuable time and resources.

Benefits of continuous deployment

The following are the key benefits of continuous deployment:

- **Enhanced safety:** Continuous deployment allows for the implementation of various deployment techniques and patterns, such as blue/green deployment. These techniques ensure that deployments are reliable and minimize the risk of downtime or disruptions. By employing robust deployment strategies, organizations can confidently release their software, knowing that they have a safety net in place.
- **Accelerated delivery:** With continuous deployment, there is no longer a need for arduous manual steps in the deployment process. Everything is automated through well-defined scripts and pipelines. This automation significantly speeds up the

delivery process, reducing human errors and enabling faster time-to-market for new features and bug fixes.

- **Cost efficiency:** By automating the deployment process, continuous deployment eliminates the need for resource-intensive manual deployment activities. Developers and operations teams can focus on their core tasks, such as writing code and maintaining infrastructure, rather than spending hours on repetitive and error-prone deployment tasks. This cost-effective approach optimizes resource allocation and improves overall team productivity.
- **Improved team collaboration:** Continuous deployment eliminates the potential for arguments or lack of confidence among team members stemming from manual deployment errors or overlooked steps. With a reliable and automated deployment process, teams can work together more harmoniously and with increased confidence. The focus can shift towards collaborative problem-solving and innovation rather than firefighting production issues.

By embracing continuous deployment, organizations can achieve safer, faster, more cost-efficient, and collaborative software delivery. It empowers teams to consistently deliver high-quality software with reduced risks, enabling them to respond rapidly to changing market needs and deliver value to customers more frequently.

Case study

In this practical case study, we will apply the concepts and techniques learned in this chapter to the project created in the previous chapter about SignalR. Our goal is to demonstrate the step-by-step implementation of the following actions:

1. **Creating a Dockerfile:** We will create a Dockerfile, which is a text file that contains instructions for building a Docker image. The Dockerfile defines the environment, dependencies, and configurations needed for our application. We will carefully craft the Dockerfile to ensure the desired runtime environment and include any necessary build steps.
2. **Building a Docker image:** Using the Dockerfile, we will build a Docker image. The image is a lightweight, portable, and self-contained package that includes everything needed

to run our application. We will follow the best practices and leverage Docker commands to build the image efficiently, considering factors such as image size, caching, and layering.

- 3. Applying CI and CD:** Once we have our Docker image, we will integrate it into a CI/CD pipeline using Azure DevOps. We will configure a build pipeline to automatically build the Docker image whenever changes are pushed to the repository. This will ensure that the image stays up to date with the latest code changes and dependencies.

Next, we will set up a release pipeline to deploy the Docker image to the desired target environment, such as a development, staging, or production environment. The release pipeline will handle the deployment process, including steps like container registry authentication, image tagging, and orchestrating the deployment to the target platform.

Throughout this practical study case, we will explore the various configuration options, settings, and best practices for building Docker images, implementing CI, and orchestrating CD using Azure DevOps. By following along with the hands-on examples, you will gain practical experience in creating Dockerfiles, building Docker images, and automating the CI/CD process for containerized applications.

The following project solution will serve as our focal point for applying the concepts and practices covered in this case study, as we can see from the figure below:

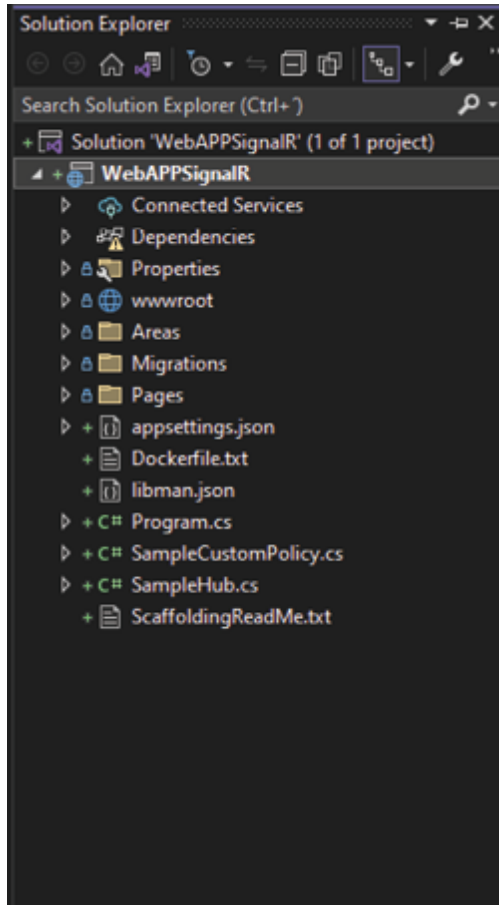


Figure 11.1: Project Solution with Dockerfile

Creating the Dockerfile

The Dockerfile must be in the same directory as the `.csproj`. The Dockerfile used in this example is the simplest one, as follows:

1. `FROM mcr.microsoft.com/dotnet/sdk:7.0 AS build-env`
2. `WORKDIR /App`
- 3.
4. `# Copy everything`
5. `COPY . ./`
6. `# Restore as distinct layers`


```

7. RUN dotnet restore
8. # Build and publish a release
9. RUN dotnet publish -c Release -o out
10.
11. # Build runtime image
12. FROM mcr.microsoft.com/dotnet/aspnet:7.0
13. WORKDIR /App
14. COPY --from=build-env /App/out .
15. ENTRYPOINT ["dotnet", "WebAPPSignalR.dll"]

```

The Dockerfile above is responsible to build the project and publish the image.

Creating the Docker image

Now that we have the Dockerfile, the next step is to create a Docker image from it. The following steps are required to create the image:

1. Open cmd.
2. Change the directory to the Dockerfile location
3. Execute the following command. This command will create an image named **webapp-thiago-image**:

```
docker build -t webapp-thiago-image -f Dockerfile .
```

This is the output from the command:

```

C:\Personal\thiago-vivas\Books\c# 12\Chapter 11\WebAPPSignalR>docker build -t webapp-thiago-image -f Dockerfile .
[*] Building 49/1s (14/54) FINISHED
-> [internal] load build definition from Dockerfile
=> transferring dockerfile: 414B
=> [internal] load .dockerignore
=> transferring context: 2B
=> [internal] load metadata for mcr.microsoft.com/dotnet/aspnet:7.0
=> [internal] load metadata for mcr.microsoft.com/dotnet/sdk:7.0
=> [build-env 1/5] FROM mcr.microsoft.com/dotnet/sdk:7.0@sha256:79cac76458f5ff6b9ca7a651d5217ca4f7325db151cae01f6712e8979
=> [internal] load build context
=> transferring context: 15.36MB
=> [stage-1 1/1] FROM mcr.microsoft.com/dotnet/aspnet:7.0@sha256:7b79949798a22682164efc191833ad38ca878d5bc25feeb238b538966c657
=> CACHED [stage-1 2/3] WORKDIR /App
=> CACHED [build-env 2/5] WORKDIR /App
=> [build-env 3/5] COPY . /
=> [build-env 4/5] RUN dotnet restore
=> [build-env 5/5] RUN dotnet publish -c Release -o out
=> [stage-1 3/3] COPY --from=build-env /App/out .
=> exporting layers
=> exporting image sha256:a8b9a387d81aba31ce7224cefb4ec5ceec3e515f66db6af9fb685d1f3775
=> naming to docker.io/library/webapp-thiago-image
Successfully built webapp-thiago-image

```

Figure 11.2: Command prompt with the output from the image creation command

Validating if the image was successfully created, we can see the output from command prompt below by executing the command:

Docker images

```
C:\WINDOWS\system32\cmd.exe

C:\Pessoal\thiago-vivas\Books\C# 12\Chapter 11\WebAPPSignalR>Docker images
REPOSITORY          TAG             IMAGE ID        CREATED         SIZE
webapp-thiago-image  latest         a8ba9a387d81   2 minutes ago   300MB

C:\Pessoal\thiago-vivas\Books\C# 12\Chapter 11\WebAPPSignalR>
```

Figure 11.3: Command prompt with the command to list all the images from the Docker

The following we can see the figures in Docker Desktop:

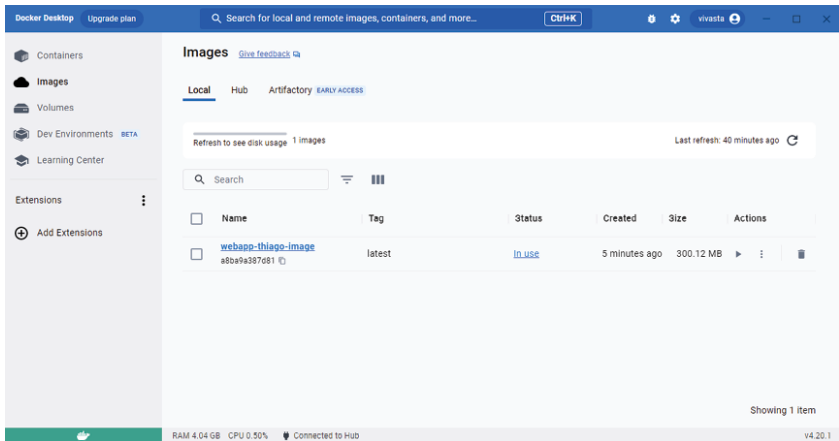


Figure 11.4: Docker desktop with the recently created image

Run the image

As we have the image created, let us run it to see if it was created successfully. The command below is going to create a Docker container, exposing the ports, with our image:

```
docker run -d -p 5000:80 webapp-thiago-image
```

And we have the following output in command prompt:

```
C:\WINDOWS\system32\cmd.exe

C:\Pessoal\thiago-vivas\Books\C# 12\Chapter 11\WebAPPSignalR>docker run -d -p 5000:80 webapp-thiago-image
bfb6cd00ebcb814082b06d4448d8eb1decc66498cdfccc71a78d6376ad98d6d

C:\Pessoal\thiago-vivas\Books\C# 12\Chapter 11\WebAPPSignalR>
```

Figure 11.5: Command prompt with the output from running the image

From Docker Desktop, we can see the container created with the image running on the specified ports:

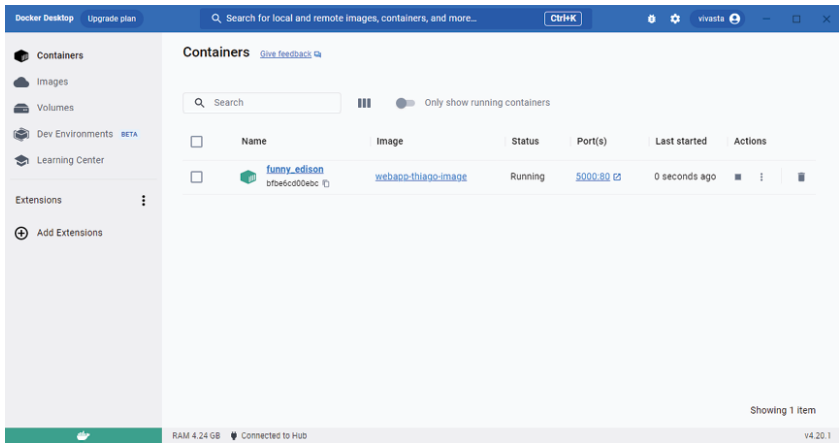


Figure 11.6: Docker desktop with the container created and image running on it
Here we have our Web APP running on the browser:

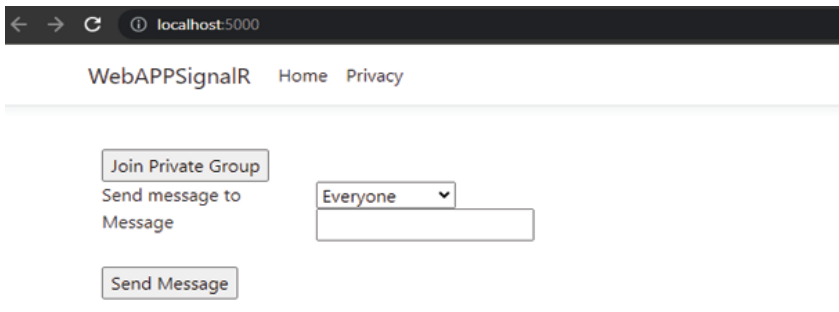


Figure 11.7: Browser running the SignalR Web APP

Applying continuous integration

As we already tested our Dockerfile and confirmed that it is working properly now, it is time to apply continuous integration to make sure that we have an updated image every time we push changes to the repository.

The continuous integration will be responsible to build the project

and push the Docker image to a container registry.

To proceed, the reader needs to upload their project to an Azure DevOps repository. This step must be completed independently before moving on to the next section.

Now, let us make sure that our project is uploaded to an Azure DevOps repository, as we can see from the image below:

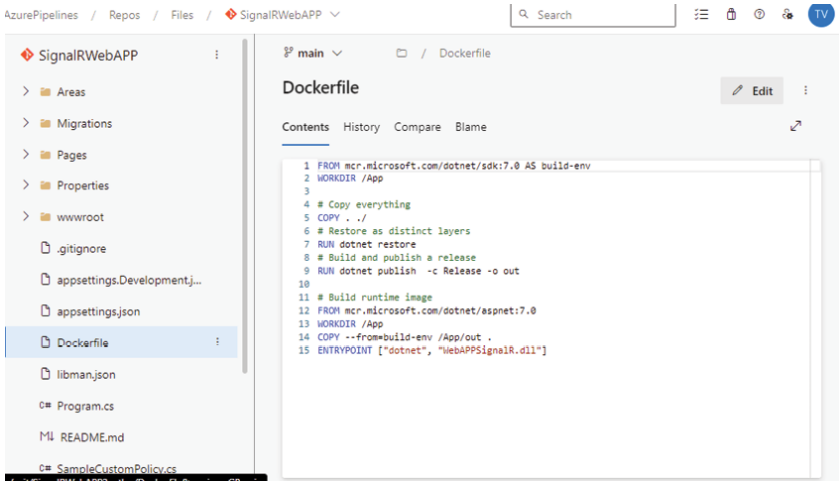


Figure 11.8: Repository with the Web APP project in Azure DevOps

Now, we will create a new pipeline:

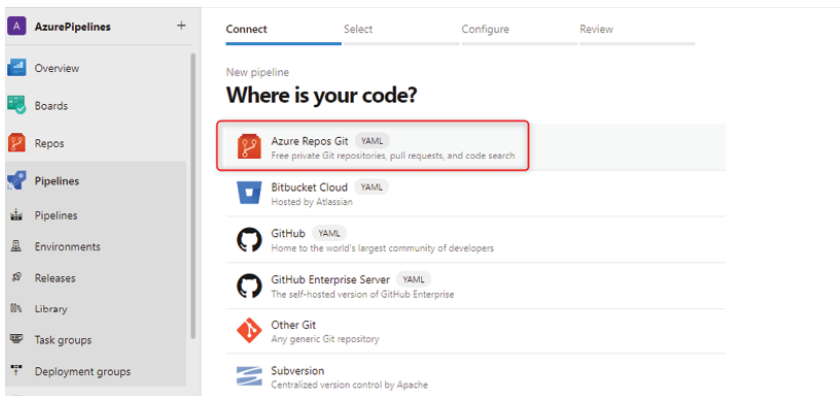


Figure 11.9: Selecting the repository where we have our code in Azure DevOps

Select the recently created repository:

Select a repository

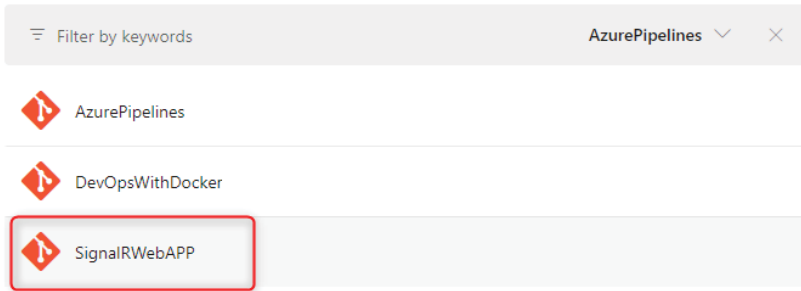


Figure 11.10: The repository to be used in this example

Start with an empty `.yaml`, paste the following code that will be the `.yaml` that we are using:

```
1. # Docker
2. # Build a Docker image
3. # https://docs.microsoft.com/azure/devops/pipelines/
   languages/docker
4.
5. trigger:
6. - main
7.
8. resources:
9. - repo: self
10.
11. variables:
12.   tag: '$(Build.BuildId)'
13. stages:
14. - stage: Build
15.   displayName: Build image
16.   jobs:
17.   - job: Build
18.     displayName: Build
19.     pool:
20.       vmImage: 'ubuntu-latest'
```

```

21.     steps:
22.     - task: Docker@2
23.     inputs:
24.         containerRegistry: 'dockerVivastaa'
25.         repository: 'vivastaa/devops'
26.         command: 'build'
27.         Dockerfile: '**/Dockerfile'
28.     - task: Docker@2
29.     inputs:
30.         containerRegistry: 'dockerVivastaa'
31.         repository: 'vivastaa/devops'
32.         command: 'logout'
33.         Dockerfile: '**/Dockerfile'
34.     - task: Docker@2
35.     inputs:
36.         containerRegistry: 'dockerVivastaa'
37.         repository: 'vivastaa/devops'
38.         command: 'login'
39.         Dockerfile: '**/Dockerfile'
40.     - task: Docker@2
41.     inputs:
42.         containerRegistry: 'dockerVivastaa'
43.         repository: 'vivastaa/devops'
44.         command: 'push'
45.         Dockerfile: '**/Dockerfile'

```

The `.yaml` above has four tasks, that are highlighted in yellow:

- The first builds the project and create the Docker image.
- This is a workaround because of authentication issues in Azure DevOps, we have to make sure that we have logged out from the container registry.
- Now, we will login again.
- We push the image created in the first step.

In the previous `.yaml` there are some references to `dockerVivastaa`, which is a service connection. It handles the connection between Azure DevOps and the Docker Container Registry used in this study

case. We can see the dockerVivastaa service connection from the image below:

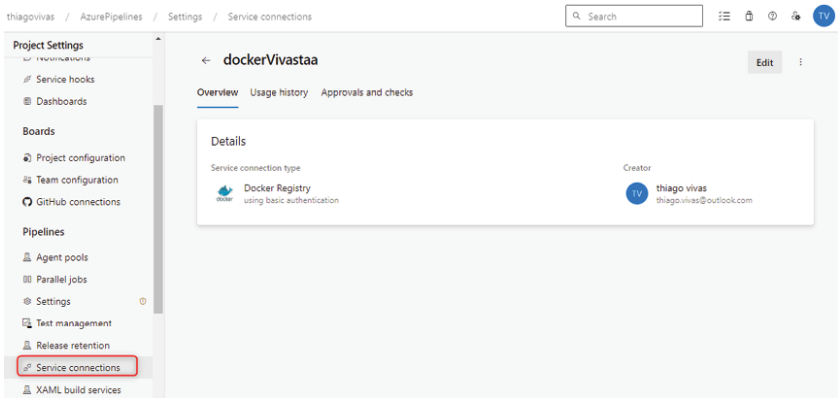


Figure 11.11: Azure DevOps Service Connection for Docker Container Registry

Following figure shows the Docker Container Registry being used:

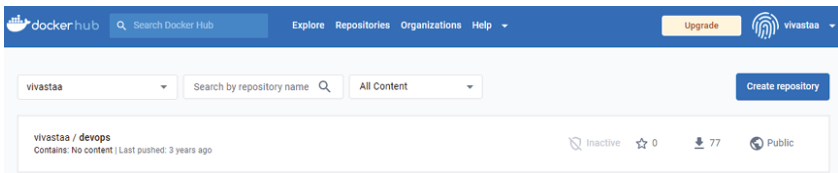


Figure 11.12: The Docker container registry used in this example

After saving the `.yaml` and running the pipeline, we have the output like in the image below.

To manually start a pipeline from the Pipelines section in Azure DevOps, follow these steps:

1. Navigate to your Azure DevOps project and select Pipelines from the left sidebar.
2. In the Pipelines section, you will see a list of all your pipelines. Find the pipeline you want to run.
3. Click on the name of the pipeline to open its details page.
4. On the pipeline details page, click the **Run pipeline** button at the top right corner.
5. A dialog box will appear, allowing you to select the branch and configure any parameters for the run.

6. After configuring the desired settings, click **Run** to start the pipeline:

The screenshot shows the Azure DevOps build system interface. On the left, a sidebar titled 'Jobs in run #20230...' lists the build steps for 'SignalRWebAPP'. The steps are: Build (1m 19s), Initialize job (<1s), Checkout Signal... (1s), Docker (1m 0s), Docker (<1s), Docker (<1s), Docker (15s), Post-job: Check... (<1s), Finalize Job (<1s), and Report build st... (<1s). The 'Post-job: Check...' step is currently selected. The main panel displays the output for this step, titled 'Post-job: Checkout SignalRWebAPP@main t...'. The output shows the task 'Get sources' with a description: 'Get sources from a repository. Supports Git, TfsVC, and SVN repositories.' The task is version 1.0.0, authored by Microsoft. The output also shows the task cleaning any cached credential from the repository: 'SignalRWebAPP (git)' and finishing the checkout process.

Step	Duration
Build	1m 19s
Initialize job	<1s
Checkout Signal...	1s
Docker	1m 0s
Docker	<1s
Docker	<1s
Docker	15s
Post-job: Check...	<1s
Finalize Job	<1s
Report build st...	<1s

```
1 Starting: Checkout SignalRWebAPP@main to s
2 =====
3 Task : Get sources
4 Description : Get sources from a repository. Supports Git, TfsVC, and SVN repositories.
5 Version : 1.0.0
6 Author : Microsoft
7 Help : [More Information](https://go.microsoft.com/fwlink/?LinkID=298199)
8 =====
9 Cleaning any cached credential from repository: SignalRWebAPP (git)
10 Finishing: Checkout SignalRWebAPP@main to s
```

Figure 11.13: The output from the build execution

Validating if the image was uploaded to our Docker container registry:

The screenshot shows the Docker Hub repository page for 'vivastaa / devops'. The page has a breadcrumb trail: 'vivastaa > Repositories > devops > General'. The 'General' tab is selected, showing a description field with a placeholder 'Add a short description for this repository'. Below this, the repository name 'vivastaa / devops' is displayed, followed by the description 'This repository does not have a description'. The 'Last pushed' time is 'a minute ago'. The 'Tags' section shows 'This repository contains 1 tag(s)'. A table lists the tags, with one tag '46' shown, which is an 'Image' type, pushed '2 minutes ago'.

Tag	OS	Type	Pulled	Pushed
46		Image	---	2 minutes ago

Figure 11.14: Docker Container Registry with the recently uploaded image

Applying continuous deployment

Now that we have our image updated every time, we push changes to the repository, it is time to publish this image. The continuous deployment process starts right after the successful continuous integration process.

The continuous deployment process will get the latest image in the Docker container registry and publish it to a Web App running on Azure.

Let us create a new release pipeline, starting from the Artifact, which is going to be our previously created build during the continuous integration process. Let us start by adding the artifact to our pipeline. Then, we add an Azure App Service Deploy task and configure it according to our settings, as we can see from the figure below:

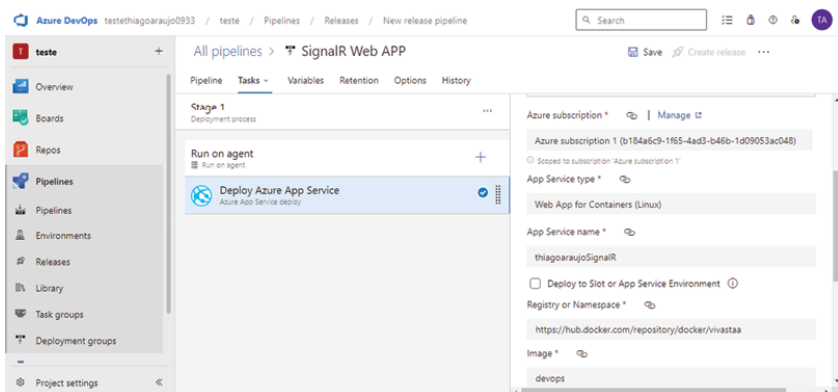


Figure 11.15: Setting the deployment task configuration

After configuring, we can deploy the pipeline, and this must be the output:

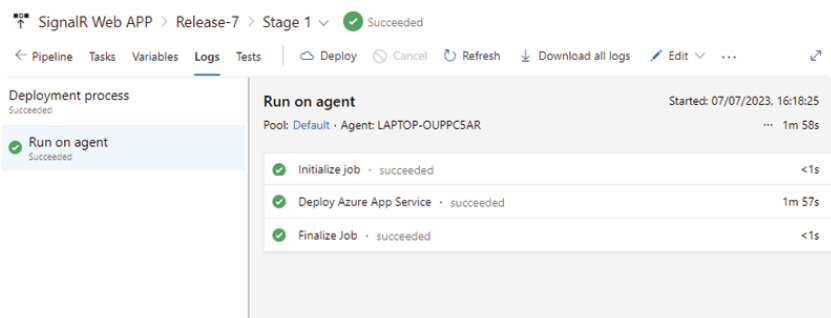


Figure 11.16: The output from the release deployment

Validating our Web App in Azure portal, it was successfully deployed as we can see from the figure below:

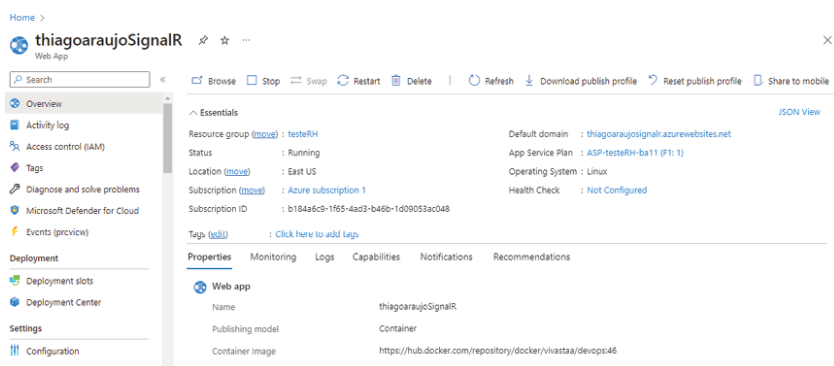


Figure 11.17: The Web App in Azure portal after the success deployment

Conclusion

In this chapter, we explored the implementation of a CI/CD pipeline for containerized applications using Docker and Azure DevOps. We began by understanding the fundamentals of Docker and its role in containerization, followed by an overview of Azure DevOps as a comprehensive DevOps platform. By leveraging Docker and Azure DevOps together, we can streamline the CI/CD process, ensuring consistent and efficient application deployment.

We covered essential Docker commands for building and deploying containerized applications, enabling us to package and distribute our applications effectively. Additionally, we discussed the concept

of CI and its significance in automating code builds and tests, improving development efficiency and software quality.

Furthermore, we delved into CD and learned how Azure DevOps automates the CD pipeline, allowing for seamless application deployment across different environments. By configuring build pipelines, release pipelines, and managing environments in Azure DevOps, we achieved a streamlined and automated CI/CD workflow.

Through a practical case study, we applied our knowledge and skills to build a sample containerized application, set up a complete CI/CD pipeline using Azure DevOps, and successfully deployed the application to production environments. This hands-on experience solidified our understanding of the CI/CD process with Docker and Azure DevOps.

By mastering the concepts and techniques covered in this chapter, you are now equipped to implement efficient and scalable CI/CD pipelines for containerized applications. Leveraging Docker and Azure DevOps, you can ensure consistent and reliable application deployment, enabling faster delivery and improved software quality. With CI/CD, you can enhance your development process, respond to changing demands swiftly, and deliver value to your users more efficiently. In the upcoming chapter, you will explore the powerful capabilities of .NET MAUI and the unique features of Blazor Hybrid. The chapter begins with a comprehensive overview of .NET MAUI, delving into its fundamental concepts and capabilities to establish a solid foundation for multi-platform app development. Discover the distinctions between Blazor and Blazor Hybrid, gaining insights into when to leverage each technology.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 12

Building Multi-platform Apps with .NET MAUI and Blazor Hybrid

Introduction

Welcome to the exciting world of building multi-platform apps with **.NET Multi-platform App UI (MAUI)** and **Blazor Hybrid**. In this chapter, we will embark on a journey that combines the power of .NET MAUI with the innovative approach of Blazor Hybrid to create cutting-edge applications that run seamlessly across various platforms.

In the first section, we will provide you with a comprehensive overview of .NET MAUI, the next evolution of Xamarin.Forms. We will explore its capabilities, advantages, and how it empowers developers to build native applications for iOS, Android, macOS, and Windows using a single codebase. Understanding the fundamentals of .NET MAUI is crucial as it sets the foundation for

the rest of our exploration.

Next, we will delve into the key differences between Blazor and Blazor Hybrid. While Blazor allows developers to build web applications using C# and .NET, Blazor Hybrid introduces a revolutionary concept that combines web technologies with native app development. By understanding these distinctions, we will gain insight into how Blazor Hybrid can enhance our cross-platform development process.

The heart of this chapter lies in our in-depth case study, where we will take you through a step-by-step implementation of a multi-platform app using .NET MAUI and Blazor Hybrid. You will learn how to create a .NET MAUI project from scratch and set up the Blazor Hybrid UI, where we will leverage web technologies to build interactive user interfaces. By following this practical example, you will gain hands-on experience and a solid understanding of the integration between .NET MAUI and Blazor Hybrid.

So, whether you are a seasoned .NET developer curious about the latest advancements or someone eager to explore the potential of cross-platform development, this chapter will equip you with the knowledge and skills to build versatile and feature-rich applications that cater to diverse platforms and audiences.

Let us dive into the world of multi-platform apps with .NET MAUI and Blazor Hybrid and unlock the possibilities of creating groundbreaking experiences for users worldwide.

Structure

This chapter covers the following topics:

- .NET MAUI overview
- Differences between Blazor X Blazor Hybrid
- Case study with step-by-step implementation
 - Creating the .NET MAUI project
 - Using Blazor Hybrid UI from Desktop client
 - Using Blazor Hybrid UI from mobile client

Objectives

In this chapter, we will discuss .NET MAUI essentials, exploring core concepts for robust cross-platform apps. Differentiate between Blazor and Blazor Hybrid, understanding the web framework and its hybrid integration for diverse platforms. Follow a step-by-step case study to create a multi-platform app, seamlessly blending .NET MAUI with Blazor Hybrid UI. Leverage web technologies like HTML, CSS, and C# within native applications for responsive interfaces. Craft cross-platform apps effortlessly on iOS, Android, macOS, and Windows from a single codebase. Uncover the power of .NET MAUI and Blazor Hybrid, enhancing development skills for innovative, feature-rich applications across varied platforms, fostering creativity and staying at the forefront of modern development.

.NET MAUI overview

.NET MAUI is a powerful framework and the next evolution of Xamarin.Forms, developed by Microsoft. It allows developers to build native applications for iOS, Android, macOS, and Windows from a single codebase. The primary goal of .NET MAUI is to simplify cross-platform app development and provide a consistent and unified development experience across various platforms.

Key features and components of .NET MAUI:

- **Cross-platform development:** With .NET MAUI, developers can write code once and deploy it on multiple platforms, saving time and effort compared to developing separate native applications for each platform.
- **Unified API:** .NET MAUI offers a unified API surface, which means developers can access platform-specific functionality through a single codebase. This simplifies the development process and allows for a seamless user experience across platforms.
- **Adaptive UI:** .NET MAUI provides adaptive UI capabilities, allowing developers to create user interfaces that automatically adapt to different screen sizes and orientations. This ensures that the app looks and functions optimally on various devices.
- **Native performance:** .NET MAUI applications run as native apps on each platform, utilizing native controls and APIs. This results in high-performance and responsive apps that provide a native look and feel to users.

- **Hot reload:** .NET MAUI supports a hot reload feature, enabling developers to see changes instantly during development without restarting the application. This accelerates the development process and improves productivity.
- **Xamarin Community Ecosystem:** As the successor to Xamarin.Forms, .NET MAUI benefits from a robust ecosystem of libraries, tools, and community support, making it easier for developers to find resources and solutions for their projects.
- **Support for .NET 6:** .NET MAUI is built on top of .NET 6, which brings new features, improvements, and optimizations to the .NET ecosystem. This ensures that developers can leverage the latest advancements in the .NET framework.

.NET MAUI aims to provide a versatile and streamlined development experience, allowing developers to reach a broader audience by targeting multiple platforms with a single codebase. Whether you are a seasoned .NET developer or new to cross-platform development, .NET MAUI offers an exciting opportunity to create modern and feature-rich applications that cater to diverse user needs on various devices.

Differences between Blazor X Blazor Hybrid

Blazor and Blazor Hybrid are two related but distinct technologies within the .NET ecosystem that enable developers to build web applications and hybrid cross-platform apps. Let us explore the key differences between Blazor and Blazor Hybrid:

	Blazor	Blazor Hybrid
Application Type	Web applications (iOS, Android, macOS, Windows)	Cross-platform apps (iOS, Android, macOS, Windows)
Rendering	Server-side rendering (SSR) using SignalR	Client-side rendering (CSR) using SignalR
UI Framework	Blazor, Razor, and Razor syntax with natives	Blazor, Razor, and Razor syntax with natives
Deployment	Web applications (SaaS) like user experiences	Web applications (SaaS) like user experiences
Integration	Integration with native code and APIs	Integration with native code and APIs
Typical Use Case	Web applications (SaaS) like user experiences	Web applications (SaaS) like user experiences
Code Sharing	Code sharing between different platforms	Code sharing between different platforms

Table 12.1: Differences between Blazor and Blazor Hybrid.

In summary, Blazor is primarily focused on building web applications that run in the browser, whereas Blazor Hybrid is designed for creating cross-platform apps that combine web

technologies with native app development. Blazor Hybrid leverages .NET MAUI to extend its reach to various platforms and provide a more native-like experience to users.

Case study with step-by-step implementation

In the world of cross-platform application development, practical experience is often the best teacher. That is why we have designed this comprehensive case study with step-by-step implementation to guide you through the process of building a multi-platform app using .NET MAUI and Blazor Hybrid.

This case study is divided into three key segments, each focusing on a crucial aspect of the development process.

Creating the .NET MAUI project

First things first, let us start with our .NET MAUI project creation, as the following figure shows:

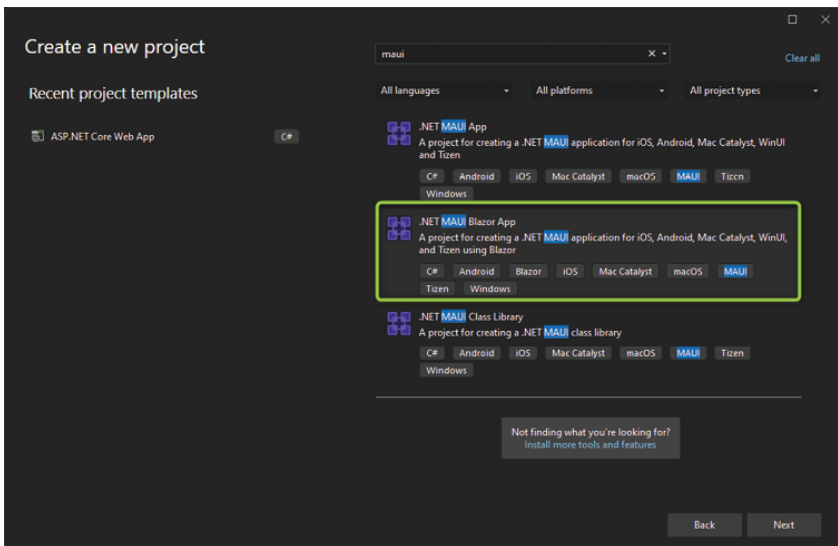


Figure 12.1: Creating a new project based on the .NET MAUI Blazor App template.

Visual Studio will provide us a project ready to be run. Our project solution must look like the picture below, now push **F5**:

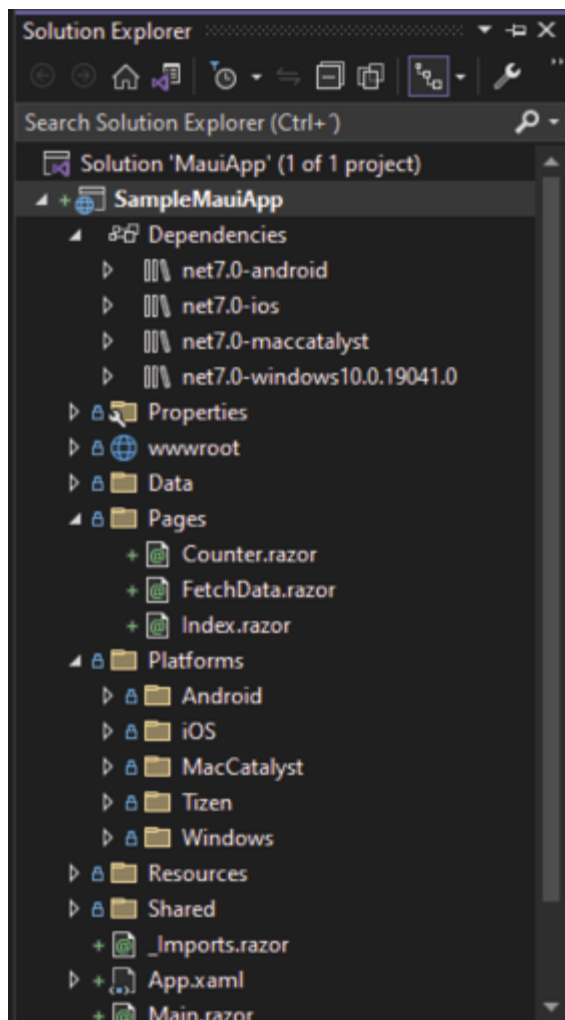


Figure 12.2: Visual Studio solution with the items from the template.

Using Blazor Hybrid UI from Desktop client

Setting up Blazor Hybrid locally is a crucial step in your journey to building cross-platform applications that seamlessly blend web technologies with native app development. This process forms the foundation for your development environment, enabling you to harness the power of Blazor Hybrid and .NET MAUI to create feature-rich, multi-platform apps that cater to a diverse audience.

In this section, we will guide you through the essential steps

required to configure your development environment for Blazor Hybrid. Whether you are an experienced developer looking to expand your skillset or a newcomer eager to explore the possibilities of cross-platform app development, this guide will provide you with the knowledge and tools to get started.

If you push *F5* for the first time, you are probably going to be prompted with the following message saying that you should set your device into developer mode:

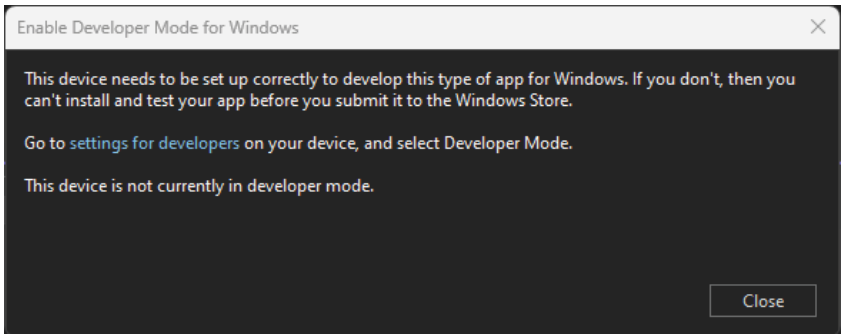


Figure 12.3: Visual Studio message saying that you should enable developer mode.

To enable the developer mode, we should go to **Privacy & security**, then enable the **Developer Mode** as follows:

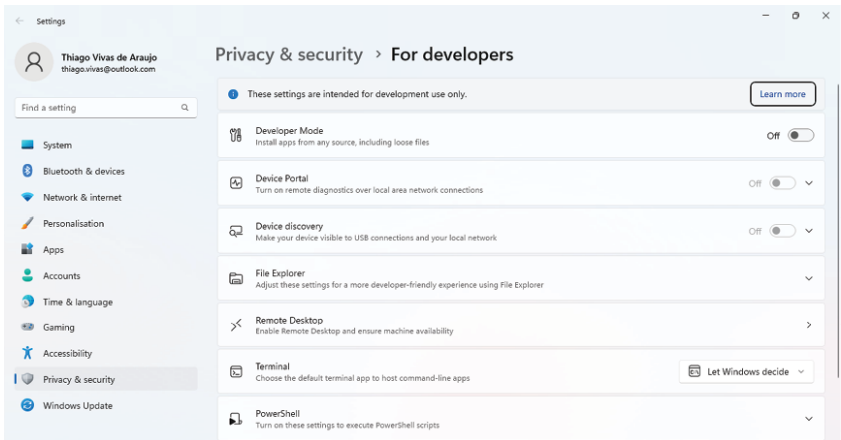


Figure 12.4: Windows Privacy & Security Control Panel.

After successfully setting up the developer mode now, we can run our project as a Windows Desktop Application, and the result is the one as follows:

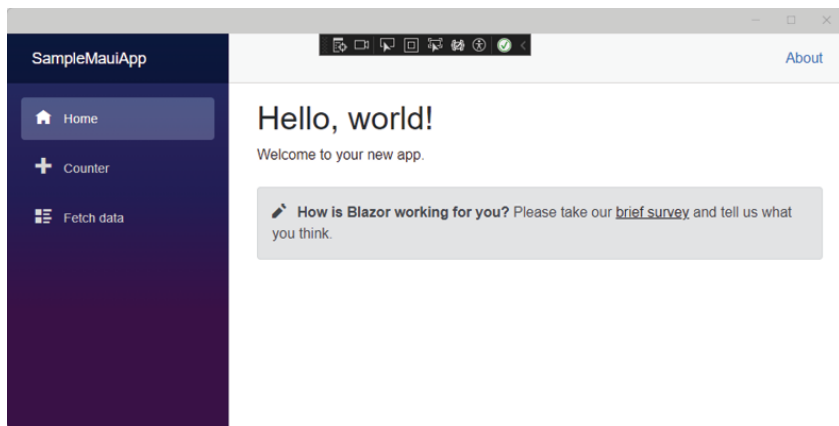


Figure 12.5: Blazor Hybrid project running as a Windows Desktop application.

Using Blazor Hybrid UI from mobile client

Once your development environment is set up, it's time to dive into the heart of Blazor Hybrid. We will provide practical example and step-by-step instructions to help you harness the power of web technologies within your Android native application, achieving a unique and dynamic user experience.

After running our project as a Windows Desktop Application, we are now going to run our project as a Mobile project using the Android Emulator, as we can see from the figure below:

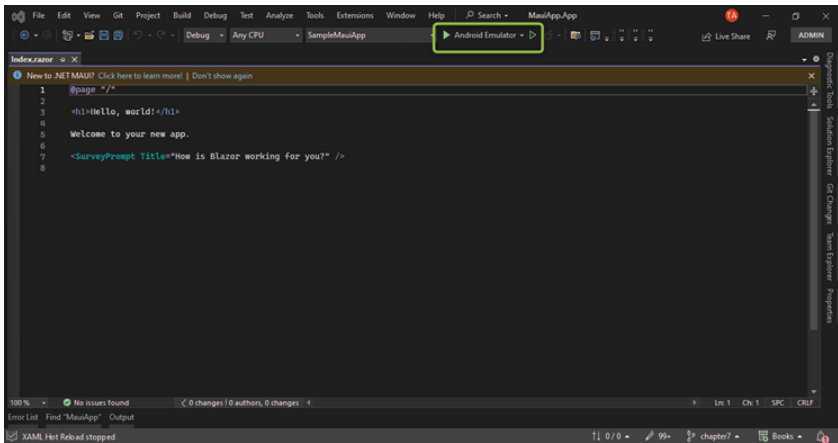


Figure 12.6: Running the project from an Android Emulator.

If this is your first time running an Android Emulator, then you would need to set it up first by creating an Android Device as follows:

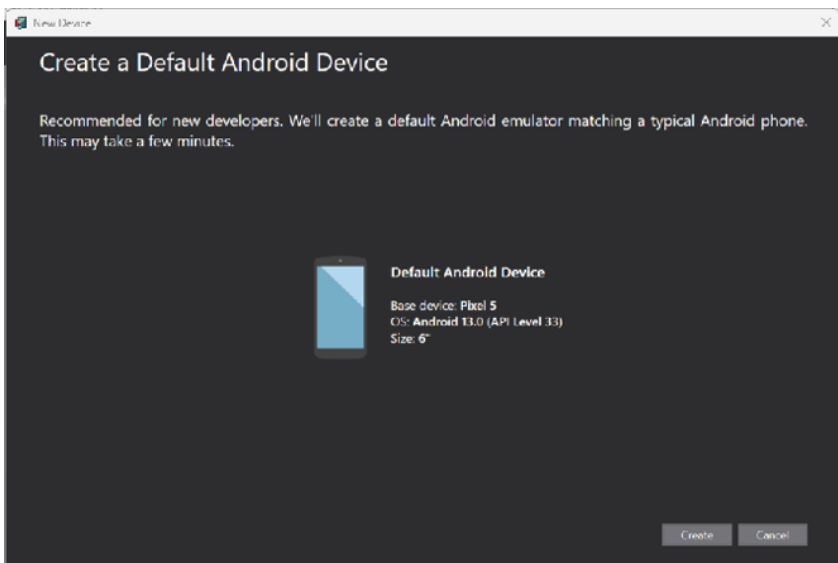


Figure 12.7: Creating an Android Device for the Android Emulator.

A new window will pop up with the emulator download as follows:

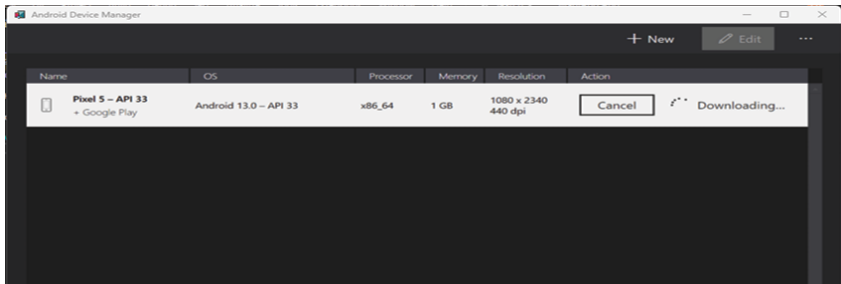


Figure 12.8: Downloading Pixel 5 Android Emulator.

After successfully setting up the Android Emulator, we can finally run our project from an Android device:

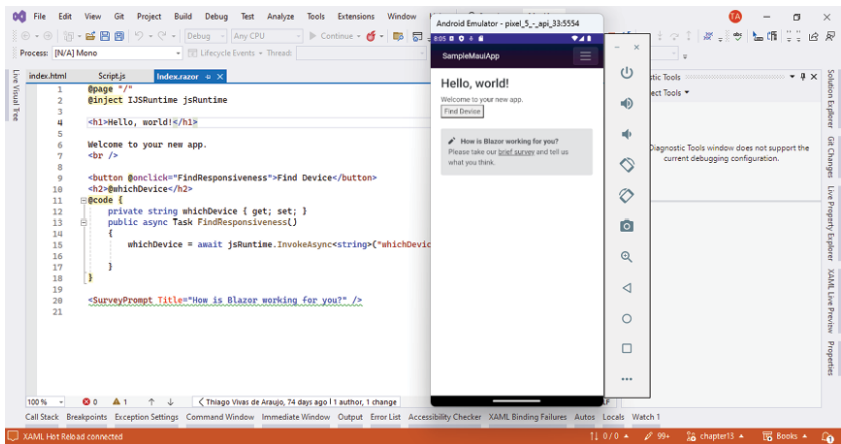


Figure 12.9: Our Blazor Hybrid project running from a Pixel 5 Android Emulator.

Note: We have not made any changes to the original project created from Visual Studio's template.

Working with Blazor Hybrid UI is an exhilarating endeavor that offers a seamless experience for developers. The beauty of this technology lies in its ability to fuse web and native app development, creating components that resemble those from web applications. The best part is that it is remarkably straightforward.

Blazor Hybrid UI empowers developers to build components that are rich in interactivity and responsive design, much like those seen in web applications. By using familiar web technologies, such as HTML, CSS, and C#, you can effortlessly craft dynamic user

interfaces, complete with event handling and real-time updates.

So, whether you are accustomed to web development or are a seasoned app developer, you will find that creating Blazor Hybrid UI components is an intuitive process. This ease of use, combined with the power of .NET MAUI, makes it an exciting platform to bring your app development ideas to life, all while providing your users with a captivating and seamless experience across multiple platforms. Let us dive into the details and discover just how accessible and versatile working with Blazor Hybrid UI can be.

Conclusion

Throughout our exploration, we began by understanding the essence of .NET MAUI and discovering how it simplifies the process of building native apps for multiple platforms. By learning the key differences between Blazor and Blazor Hybrid, we gained valuable insights into the potential of integrating web technologies with native app development.

The highlight of our chapter was the comprehensive case study, where we took you through a practical implementation of a multi-platform app using .NET MAUI and Blazor Hybrid. By following the step-by-step instructions, you've experienced firsthand the power of combining these technologies to create interactive and versatile user interfaces that run seamlessly on various devices.

As you continue your journey as a developer, remember that the world of technology is ever-evolving. Staying curious and continuously learning is essential to stay ahead in the rapidly changing landscape of cross-platform app development.

We hope you feel inspired to experiment further with .NET MAUI and Blazor Hybrid, incorporating these cutting-edge technologies into your own projects and exploring their potential to revolutionize the way we build applications.

Thank you for joining us on this adventure! We wish you success and fulfillment as you create remarkable multi-platform apps that make a positive impact on the lives of users worldwide.

In the upcoming chapter, *Introducing WinUI, the native UX for Windows Desktop and UWP Apps*, we will discuss the fundamentals of WinUI, Microsoft's native **user experience (UX)** framework for

Windows Desktop and **Universal Windows Platform (UWP)** applications. The overview provides insights into WinUI's role in enhancing the visual and interactive aspects of applications on the Windows platform. Topics covered include the core concepts of WinUI, its integration with Windows development, and its capabilities for creating modern and responsive user interfaces. The chapter aims to equip developers with a foundational understanding of WinUI, empowering them to leverage this framework for building immersive and user-friendly Windows applications.

CHAPTER 13

Windows UI Library: Crafting Native Windows Experience

Introduction

In this chapter, we will discuss about **Windows UI Library (WinUI)**, the native **User Experience (UX)** framework for Windows Desktop and **Universal Windows Platform (UWP)** apps. We will explore the powerful and visually appealing capabilities of WinUI, as well as the elegance of the Fluent Design System, Microsoft's design language for creating modern applications.

With WinUI, developers can build native Windows applications that seamlessly blend into the user's environment, delivering a consistent and immersive experience across different devices and form factors. Whether you are developing for traditional desktop computers or cutting-edge UWP devices, WinUI provides the tools and components you need to create applications that feel right at home on the Windows platform.

We will begin with an overview of WinUI, understanding its

significance as the native UX for Windows apps and how it enhances the overall user experience. Then, we will dive into the Fluent Design System, which serves as the foundation for crafting beautiful and intuitive interfaces. By following a step-by-step case study, you will witness the transformation of a concept into a fully functional WinUI-powered application, learning key implementation techniques along the way.

Let us discuss about WinUI, where we will discover the art of designing and developing stunning Windows applications that captivate users and leave a lasting impression.

Structure

This chapter covers the following topics:

- Introducing WinUI, the native UX for Windows Desktop and UWP Apps
- Fluent Design System
- Case study with step-by-step implementation
 - Creating the project
 - Design the UI
 - Implementing the cache
 - Data transfer between pages

Objectives

In this chapter, you will grasp WinUI's importance in crafting native UX for Windows apps, exploring the Fluent Design System and its principles. Follow a step-by-step case study to build a real app, from setup to deployment. Learn to design engaging UIs and implement app logic seamlessly. By chapter's end, you will confidently deploy WinUI apps, ready to create delightful user experiences on Windows Desktop and UWP platforms, showcasing expertise in modern app development.

Windows UI Library Introduction

WinUI is a modern native UX framework developed by Microsoft. It serves as the primary UI framework for building native Windows

applications, supporting both Windows Desktop and UWP apps. WinUI empowers developers to create visually appealing, responsive, and highly performant applications that seamlessly adapt to various Windows devices and form factors.

Key features and advantages of WinUI:

- **Native Windows integration:** WinUI is deeply integrated with the Windows operating system, providing direct access to the latest Windows features, APIs, and user experiences. This ensures that applications built with WinUI feel like a natural part of the Windows ecosystem.
- **Consistent design language:** With the Fluent Design System as its core, WinUI enables developers to implement a consistent and visually engaging user interface. The Fluent Design System offers a set of design principles and components that create a smooth and immersive experience across different devices and platforms.
- **Modularity and compatibility:** WinUI is designed to be modular, allowing developers to use only the components they need, reducing the app's size and ensuring better performance. It is backward compatible, making it easier to upgrade existing apps and leverage the latest features.
- **Performance and responsiveness:** WinUI is optimized for performance, ensuring fast rendering and smooth animations, resulting in a responsive and interactive user experience.
- **Open source:** Microsoft has embraced open-source development for WinUI, which allows the community to contribute to its improvement actively. This fosters innovation, faster bug fixes, and feature enhancements.
- **Support for both XAML and C++:** WinUI supports both XAML and C++, catering to developers with different programming backgrounds and preferences. XAML enables a declarative approach to UI design, while C++ allows for more fine-grained control and performance optimizations.
- **Windows app ecosystem:** WinUI enables developers to build applications that run across a wide range of Windows devices, including desktops, laptops, tablets, 2-in-1s, Xbox, and Surface Hub. It is an essential part of the **Universal Windows Platform (UWP)**, providing a consistent development model for Windows apps.

- **Regular updates and improvements:** As part of Microsoft's commitment to delivering the best developer experience, WinUI receives regular updates and improvements, ensuring that developers can stay up-to-date with the latest advancements in the Windows platform.

Whether you are creating a traditional Windows Desktop application or targeting the latest UWP devices, WinUI empowers you to build modern and visually stunning applications that integrate seamlessly with the Windows ecosystem. By leveraging the Fluent Design System and taking advantage of WinUI's performance optimizations, you can deliver a user experience that delights users and meets the high standards set by the Windows platform.

Fluent Design System

The **Fluent Design System** is Microsoft's comprehensive design language aimed at creating visually appealing, intuitive, and engaging user experiences across a wide range of devices and platforms. Introduced in 2017, the Fluent Design System was previously known as **Microsoft Design Language 2** and has since evolved to become a significant part of the Windows user interface and application design.

Key principles of Fluent Design System

The following are the key principles of the Fluent Design System:

- **Light, depth, and motion:** The Fluent Design System uses light, depth, and motion to create a more immersive and dynamic user experience. Through the clever use of lighting and shadows, elements appear to be more tangible and interactive, while animations and transitions add a sense of movement and liveliness.
- **Material:** The Fluent Design System introduces **material** as a fundamental building block of the user interface. Materials have distinct visual properties, such as light, depth, and texture, allowing designers and developers to create cohesive and consistent UI elements.
- **Motion (Purposeful animations):** Motion plays a crucial role in the Fluent Design System, as animations serve not only to enhance visual appeal but also to provide context and feedback to users. Purposeful animations help users

understand transitions and interactions better, making the UI more intuitive.

- **Scale and consistency:** Fluent Design emphasizes scalability and consistency across devices and platforms. Whether an application runs on a desktop, tablet, or smartphone, the design language ensures a consistent user experience, adapting to different screen sizes and input methods.
- **Acrylic: Transparency and Layers:** Acrylic is a design component in the Fluent Design System that focuses on transparency and layering effects. It allows developers to create a semi-transparent background layer, adding depth and dimension to the UI while still maintaining readability and clarity.
- **Reveal highlight:** Reveal highlight is another key element of Fluent Design, offering visual feedback when interacting with UI elements. When users hover over or interact with an element, it responds with a subtle highlight, making interactions more delightful and responsive.
- **Connected animations:** Connected animations enable seamless transitions between different UI states and elements. For instance, when navigating between pages or panels, connected animations maintain visual continuity, enhancing the overall user experience.

Applications of the Fluent Design System

The Fluent Design System is primarily used for designing and developing applications on Windows, including **Universal Windows Platform (UWP)** apps and Windows Desktop applications. By leveraging the Fluent Design System's principles and components, developers can create visually stunning, consistent, and user-friendly applications that align with the overall Windows ecosystem.

Overall, the Fluent Design System allows developers and designers to craft modern, immersive, and engaging user interfaces that capture users' attention, increase usability, and elevate the overall quality of the Windows app ecosystem. As Microsoft continues to evolve the Fluent Design System, it remains an essential tool for developers to build applications that provide delightful experiences for users across a broad range of Windows devices.

Case study with step-by-step implementation

In this section, we will walk through a practical case study, demonstrating the step-by-step implementation of a Windows application using WinUI and the Fluent Design System.

Creating the project

Set up a new project to leverage WinUI for building a native Windows application.

1. Open Visual Studio and select **"Create a new project."**
2. Choose the Blank App (Universal Windows) project template for your application:

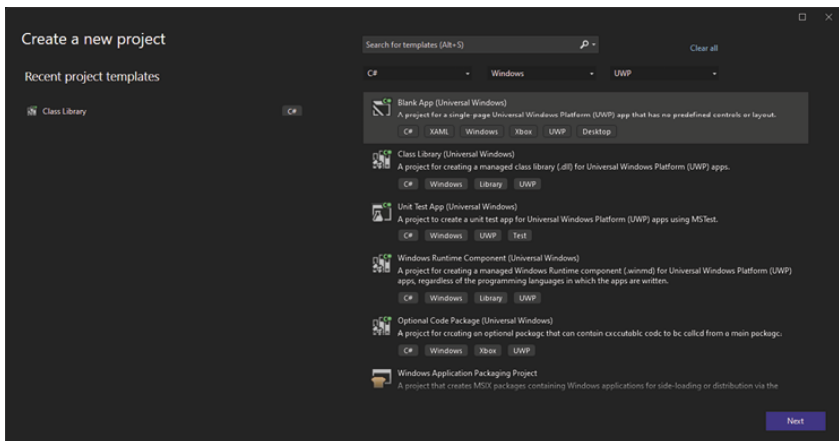


Figure 13.1: The WinUI Project template

1. Ensure that WinUI is selected as the UI framework.
2. Configure project settings, such as project name, location, and target platform.
3. Your project solution must look like this:

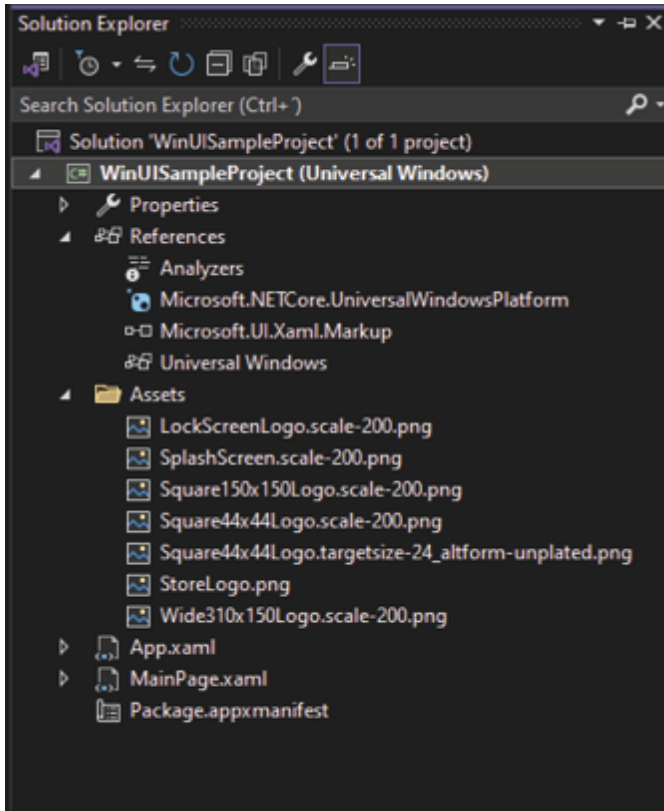


Figure 13.2: WinUI Blank Project Solution

1. Install nuget package:

- **Microsoft.UI.Xaml**

Designing the user interface

Create a visually appealing and intuitive user interface using the principles of the Fluent Design System.

Let us start with the Connected Animation transition. Add 2 different images to your project solution and name them as **SourceImage** and **DestinationImage**. We also need to add a new page and here we are calling this new page as **DestinationAnimation**.

Add the following code to the **MainPage.xaml**. This code sets the image properties and its **PointerPressed** event:

1. `<TextBlock HorizontalAlignment="Center"`

```

Text="Navigation animation"/>
2. <Image x:Name="SourceImage"
3. HorizontalAlignment="Center"
   VerticalAlignment="Top"
4. Width="200" Height="200"
5. Stretch="Fill"
6. Source="/Images/SourceImage.png"
7.
   PointerPressed="SourceImage_PointerPressed"/
>

```

Now, we will add the following code to the **MainPage.xaml.cs**. This code handles the Event and also the animation:

```

1. private void
   SourceImage_PointerPressed(object sender,
   PointerRoutedEventArgs e)
2. {
3.
   Frame.Navigate(typeof(DestinationAnimation),
   null,
4.
       new
   SuppressNavigationTransitionInfo());
5. }
6. protected override void
   OnNavigatingFrom(NavigatingCancelEventArgs
   e)
7. {
8.
   ConnectedAnimationService.GetForCurrentView()
9.
   .PrepareToAnimate("forwardAnimation",
   SourceImage);
10.
11. }

```

In our destination page, we also need to add the following block of codes. First, we must update the **DestinationAnimation.xaml** to add the image properties as follows:


```

1.     <Grid>
2.         <Image x:Name="DestinationImage"
3.             Width="400" Height="400"
4.             Stretch="Fill"
5.             Source="/Images/
DestinationImage.png"/>
6.     </Grid>

```

We also need to update the **DestinationAnimation.xaml.cs**, to receive the object and set the animation, as we can see from the following code:

```

1.     protected override void
    OnNavigatedTo(NavigationEventArgs e)
2.     {
3.         base.OnNavigatedTo(e);
4.
5.         ConnectedAnimation animation =
6.             ConnectedAnimationService.GetForCurrentView()
7.             .GetAnimation("forwardAnimation");
8.         if (animation != null)
9.         {
10.             animation.TryStart(DestinationImage);
11.         }

```

When we push *F5* to run the project and by clicking on the image we can see the animation happening.

Implementing the cache

Caching in the context of Windows applications typically refers to the practice of storing frequently accessed or computed data in a temporary storage space. This is done to enhance performance and responsiveness by avoiding redundant computations or data fetch operations.

In a WinUI application, you might implement caching in various scenarios:

- **Data retrieval:** If your application fetches data from a remote server, you could cache the retrieved data locally. Subsequent requests for the same data can then be served from the cache instead of making a new network request.
- **UI element state:** If your application involves complex UI elements or views, you might cache the state of these elements. This can be particularly useful when navigating between different pages or views.
- **Resource loading:** Caching can be applied to frequently used resources like images or styles, ensuring that they are loaded quickly when needed.
- **Performance optimization:** By caching intermediate results of computations or data processing, you can reduce the need for recalculating the same results repeatedly.

While WinUI itself may not provide a specific caching API, you can implement caching in your application using standard C# and .NET mechanisms. For example, you might use in-memory caching, local storage, or leverage third-party caching libraries that suit your application's requirements.

Always consider factors such as cache expiration, memory management, and the specific needs of your application when implementing caching.

For caching Universal Windows Applications, we must set the cache property in the page constructor as follows:

```
1. public MainPage()  
2. {  
3.     this.NavigationCacheMode =  
       NavigationCacheMode.Enabled;  
4.     this.InitializeComponent();  
5. }
```

Those are the cache options with their descriptions:

	Definition	
Display	Page is never cached and a new instance of the page is created on each visit.	
Partial	Page is cached, but the cached instance is discarded when the size of the cache for the frame is exceeded.	
Required	Page is cached and the cached instance is reused for every visit regardless of the cache size for the frame.	

Table 13.1: Navigation cache mode options

Data transfer between pages

Facilitate smooth data transition between different pages of the application.

We are creating a simple form and sending its form data to another page to show a personalized content based on the user data.

First, we create a new page and here we are naming it as **NewPage**. Then we have to add the following code to the **MainPage.xaml**. This code creates a **RadioButton** component, some text boxes and a button with a click event associated with:

```
1. <TextBlock HorizontalAlignment="Center"
   Text="Happiness form"/>
2. <TextBlock HorizontalAlignment="Center"
   Text="Enter your name"/>
3.   <TextBox
   muxc:BackdropMaterial.ApplyToRootOrPageBackground
4.       HorizontalAlignment="Center"
   Width="200" x:Name="name"/>
5.   <muxc:RadioButtons x:Name="feelings"
   HorizontalAlignment="Center">
6.       <muxc:RadioButtons.Header>
7.           <StackPanel
   Orientation="Horizontal">
8.               <SymbolIcon
   Symbol="Highlight"/>
9.               <TextBlock Text="How
   are you feeling today?"
10.                  Margin="8,0,0,0"/>
11.           </StackPanel>
12.       </muxc:RadioButtons.Header>
13.       <x:String>Good</x:String>
14.       <x:String>Very good</x:String>
15.       <x:String>Amazing</x:String>
16.   </muxc:RadioButtons>
17.   <HyperlinkButton
```

```

muxc:AnimatedIcon.State="normal"
Content="OK"
18. Click="HyperlinkButton_Click"
19. HorizontalAlignment="Center"/>

```

In the **MainPage.xaml.cs** we should add the following code to handle the button click and send the form data to our new page:

```

1. private void HyperlinkButton_Click(object
sender, RoutedEventArgs e)
2. {
3.     PayloadDTO payloadDTO = new
PayloadDTO();
4.     if (!
string.IsNullOrEmpty(name.Text))
5.         payloadDTO.Name = name.Text;
6.         if (feelings.SelectedItem != null)
7.             payloadDTO.Feel =
feelings.SelectedItem.ToString();
8.
9.     Frame.Navigate(typeof(NewPage),
payloadDTO);
10. }

```

Now we must handle the data retrieval in our **NewPage**. Let us start with the data display by adding the following code to the **NewPage.xaml**. This code has 2 text box for data display and a button to go back to the previous page:

```

1. <StackPanel
VerticalAlignment="Center">
2.     <TextBlock x:Name="greeting"
3.
HorizontalAlignment="Center"/>
4.
5.     <TextBlock x:Name="dateAndTime"
6.
HorizontalAlignment="Center"/>
7.

```

```

8.         <HyperlinkButton Content="Click to
           go back"
9.         Click="HyperlinkButton_Click"
10.        HorizontalAlignment="Center"/>
11.     </StackPanel>

```

Now, we must update the **NewPage.xaml.cs** to receive the data and display it. Also, we should handle the button event click. For this, we must insert the following block of code:

```

1. private void HyperlinkButton_Click(object
   sender, RoutedEventArgs e)
2. {
3.     Frame.Navigate(typeof(MainPage));
4. }
5. protected override void
   OnNavigatedTo(NavigationEventArgs e)
6. {
7.     if (e.Parameter is PayloadDTO &&
8.         ((PayloadDTO)e.Parameter).Name !=
   null
9.         && ((PayloadDTO)e.Parameter).Feel
   != null)
10.    {
11.        var payload =
   (PayloadDTO)e.Parameter;
12.        greeting.Text = $"Hello,
   {payload.Name}. We are happy to
13.        know that you are feeling
   {payload.Feel}";
14.    }
15.    else
16.    {
17.        greeting.Text = "Hello!";
18.    }
19.

```

```

20.     dateAndTime.Text = $"Today is
    {DateTime.UtcNow}";
21.
22.     base.OnNavigatedTo(e);
23. }

```

The following will be the result, the main page:

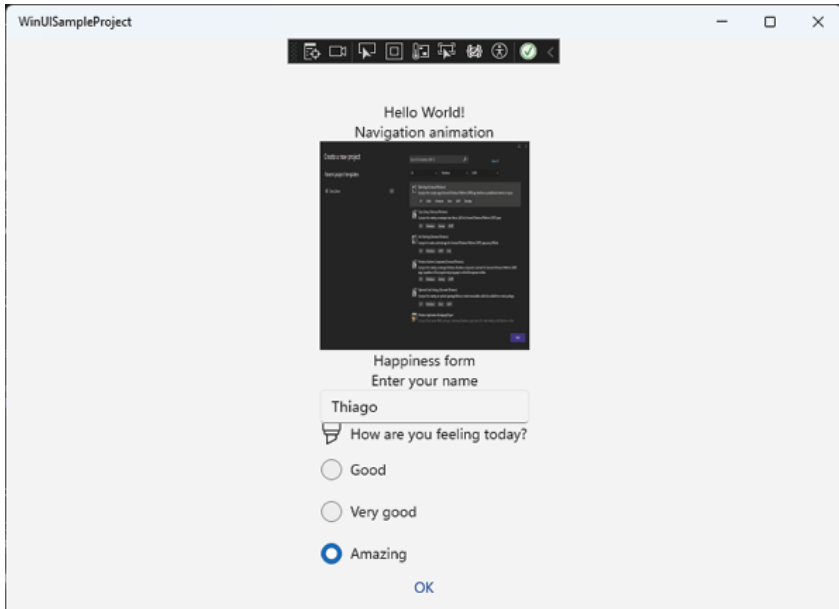


Figure 13.3: Main Page form data

The following is the new page:

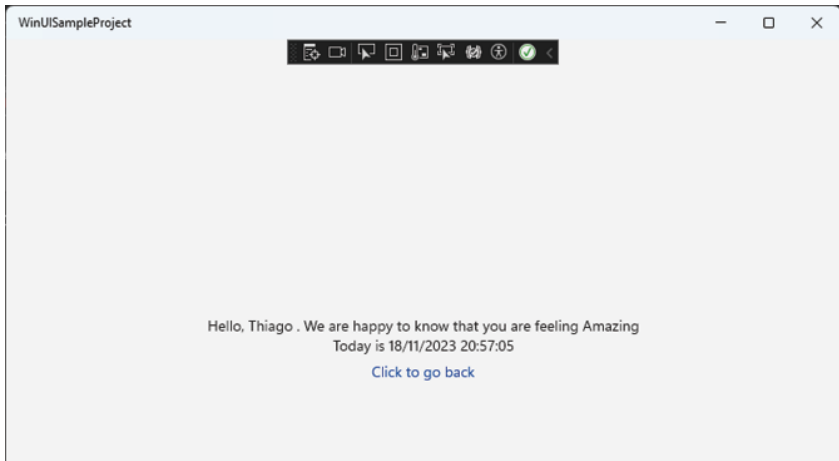


Figure 13.4: The new page with data displayed.

Following are the source codes for each page:

The main page UI source code:

MainPage.xaml:

```

1. <Page
2.     x:Class="WinUISampleProject.MainPage"
3.     xmlns="http://schemas.microsoft.com/
winfx/2006/xaml/presentation"
4.     xmlns:x="http://schemas.microsoft.com/
winfx/2006/xaml"
5.     xmlns:local="using:WinUISampleProject"
6.
xmlns:muxc="using:Microsoft.UI.Xaml.Controls"
7.     xmlns:d="http://schemas.microsoft.com/
expression/blend/2008"
8.     xmlns:mc="http://
schemas.openxmlformats.org/markup-
compatibility/2006"
9.     mc:Ignorable="d"
10.     Background="{ThemeResource
ApplicationPageBackgroundThemeBrush}"
11.     NavigationCacheMode="Enabled">
12.

```

```

13.     <StackPanel VerticalAlignment="Center">
14.         <TextBlock
            HorizontalAlignment="Center" Text="Hello
            World!"/>
15.
16.         <TextBlock
            HorizontalAlignment="Center"
            Text="Navigation animation"/>
17.         <Image x:Name="SourceImage"
18.             HorizontalAlignment="Center"
            VerticalAlignment="Top"
19.             Width="200" Height="200"
20.             Stretch="Fill"
21.             Source="/Images/SourceImage.png"
22.             PointerPressed="SourceImage_PointerPressed"/
            >
23.
24.         <TextBlock
            HorizontalAlignment="Center"
            Text="Happiness form"/>
25.         <TextBlock
            HorizontalAlignment="Center" Text="Enter
            your name"/>
26.         <TextBox
27.
            muxc:BackdropMaterial.ApplyToRootOrPageBackground
28.             HorizontalAlignment="Center"
            Width="200" x:Name="name"/>
29.         <muxc:RadioButtons
            x:Name="feelings"
30.             HorizontalAlignment="Center">
31.             <muxc:RadioButtons.Header>
32.                 <StackPanel
                    Orientation="Horizontal">
33.                     <SymbolIcon
                        Symbol="Highlight"/>

```



```

34.         <TextBlock Text="How
are you feeling today?"
35.             Margin="8,0,0,0"/>
36.     </StackPanel>
37. </muxc:RadioButtons.Header>
38.     <x:String>Good</x:String>
39.     <x:String>Very good</x:String>
40.     <x:String>Amazing</x:String>
41. </muxc:RadioButtons>
42.     <HyperlinkButton
muxc:AnimatedIcon.State="normal"
43.         Content="OK"
Click="HyperlinkButton_Click"
44.         HorizontalAlignment="Center"/>
45. </StackPanel>
46. </Page>

```

The main page C# code, setting the cache mode and configuring all the events:

MainPage.xaml.cs:

```

1. public sealed partial class MainPage :
Page
2. {
3.     public MainPage()
4.     {
5.         this.NavigationCacheMode =
NavigationCacheMode.Enabled;
6.         this.InitializeComponent();
7.     }
8.     private void HyperlinkButton_Click(object
sender, RoutedEventArgs e)
9.     {
10.         PayloadDTO payloadDTO = new
PayloadDTO();
11.         if (!
string.IsNullOrEmpty(name.Text))

```

```

12.         payloadDTO.Name = name.Text;
13.         if (feelings.SelectedItem != null)
14.             payloadDTO.Feel =
                feelings.SelectedItem.ToString();
15.
16.         Frame.Navigate(typeof(NewPage),
                payloadDTO);
17.     }
18.     private void
        SourceImage_PointerPressed(object sender,
19.         PointerRoutedEventArgs e)
20.     {
21.         Frame.Navigate(typeof(DestinationAnimation),
                null,
22.             new
                SuppressNavigationTransitionInfo());
23.     }
24.     protected override void
        OnNavigatingFrom(
25.         NavigatingCancelEventArgs e)
26.     {
27.         ConnectedAnimationService.GetForCurrentView()
28.             .PrepareToAnimate("forwardAnimation",
                SourceImage);
29.
30.     }
31. }

```

The new page UI code, with the design and form components:

NewPage.xaml:

```

<Page
    x:Class="WinUISampleProject.NewPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/
        xaml/presentation"

```

```

xmlns:x="http://schemas.microsoft.com/
winfx/2006/xaml"
xmlns:local="using:WinUISampleProject"
xmlns:muxc="using:Microsoft.UI.Xaml.Controls"
xmlns:d="http://schemas.microsoft.com/
expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/
markup-compatibility/2006"
mc:Ignorable="d"
Background="{ThemeResource
ApplicationPageBackgroundThemeBrush}"
NavigationCacheMode="Enabled">

<StackPanel VerticalAlignment="Center">
    <TextBlock x:Name="greeting"
        HorizontalAlignment="Center"/>

    <TextBlock x:Name="dateAndTime"
        HorizontalAlignment="Center"/>

    <HyperlinkButton Content="Click to go back"
        Click="HyperlinkButton_Click"
        HorizontalAlignment="Center"/>
</StackPanel>
</Page>

```

The new page C# code, with the button event:

NewPage.xaml.cs:

```

1. public sealed partial class NewPage : Page
2. {
3.     public NewPage()
4.     {
5.         this.InitializeComponent();
6.     }
7.     private void HyperlinkButton_Click(object
sender, RoutedEventArgs e)

```

```

8.      {
9.          Frame.Navigate(typeof(MainPage));
10.     }
11.     protected override void
OnNavigatedTo(NavigationEventArgs e)
12.     {
13.         if (e.Parameter is PayloadDTO &&
((PayloadDTO)e.Parameter)
14.             .Name != null
15.             &&
((PayloadDTO)e.Parameter).Feel != null)
16.         {
17.             var payload =
(PayloadDTO)e.Parameter;
18.             greeting.Text = $"Hello,
{payload.Name}. We are happy
19.             to know that you are
feeling {payload.Feel}";
20.         }
21.         else
22.         {
23.             greeting.Text = "Hello!";
24.         }
25.
26.         dateAndTime.Text = $"Today is
{DateTime.UtcNow}";
27.
28.         base.OnNavigatedTo(e);
29.     }
30. }

```

Below we can see the animation page, with a grid and our image inside it:

DestinationAnimation.xaml:

```

1. <Page
2.     x:Class="WinUISampleProject.DestinationAnimation

```

```

3.     xmlns="http://schemas.microsoft.com/
winfx/2006/xaml/presentation"
4.     xmlns:x="http://schemas.microsoft.com/
winfx/2006/xaml"
5.     xmlns:local="using:WinUISampleProject"
6.
    xmlns:muxc="using:Microsoft.UI.Xaml.Controls"
7.     xmlns:d="http://schemas.microsoft.com/
expression/blend/2008"
8.     xmlns:mc="http://
schemas.openxmlformats.org/markup-
compatibility/2006"
9.     mc:Ignorable="d"
10.     Background="{ThemeResource
ApplicationPageBackgroundThemeBrush}">
11.
12.         <Grid>
13.             <Image x:Name="DestinationImage"
14.                 Width="400" Height="400"
15.                 Stretch="Fill"
16.                 Source="/Images/
DestinationImage.png"/>
17.         </Grid>
18. </Page>

```

Below we can see the animation code in C#, configuring the **OnNavigatedTo** event: **DestinationAnimation.xaml.cs**:

```

1. public sealed partial class
DestinationAnimation : Page
2. {
3.     public DestinationAnimation()
4.     {
5.         this.InitializeComponent();
6.     }
7.     protected override void
OnNavigatedTo(NavigationEventArgs e)
8.     {

```

```

9.         base.OnNavigatedTo(e);
10.
11.         ConnectedAnimation animation =
12.
13.             ConnectedAnimationService.GetForCurrentView()
14.             .GetAnimation("forwardAnimation");
15.         if (animation != null)
16.         {
17.             animation.TryStart(DestinationImage);
18.         }
19.     }

```

You may access this project on our [GitHub repository](#).

Conclusion

We have reached the end of our exploration into WinUI, the native UX for Windows Desktop and UWP Apps. We hope this chapter has provided you with valuable insights into the power and potential of WinUI, empowering you to create immersive and visually stunning applications for the Windows platform.

Throughout this chapter, we covered the foundational concepts of WinUI and how it acts as the bridge between traditional Windows Desktop applications and the modern Universal Windows Platform. By leveraging the Fluent Design System, you now have the tools to design user interfaces that are not only aesthetically pleasing but also intuitive and user-friendly.

The step-by-step case study allowed you to witness the practical implementation of WinUI in action, taking a concept from the drawing board to a fully functional application. Armed with this knowledge, you are well-equipped to embark on your own projects, delivering delightful experiences to users on various Windows devices.

As you continue your journey as a Windows app developer, remember to stay updated with the latest advancements in the

WinUI framework and the Fluent Design System. The landscape of UX design is ever-evolving, and keeping abreast of new features and best practices will ensure that your applications remain at the forefront of innovation.

Thank you for joining us in this exploration of WinUI and the Fluent Design System. We look forward to seeing the incredible applications you will create, enriching the Windows ecosystem with your creativity and expertise.

In the upcoming chapter, we will discuss essential practices for ensuring the robustness and reliability of your code. Covering topics such as Unit Testing with NUnit and Xunit, you will learn how to systematically validate individual units of your code for correctness. Additionally, we will explore the usage of Mocks to simulate dependencies for more effective testing. Moreover, you will master debugging techniques to efficiently identify and resolve issues within your codebase. This chapter equips you with the necessary skills to enhance code quality and streamline the development process.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 14

Unit Testing and Debugging

Introduction

In the world of software development, testing and debugging are crucial aspects of ensuring the reliability and stability of our code. Unit testing allows us to validate the individual units of our code in isolation, ensuring that each component works as expected.

Additionally, debugging is an essential skill that helps us identify and resolve issues when our code does not behave as intended.

Throughout this chapter, we will explore the fundamentals of unit testing using xUnit, a popular testing framework that provides a robust and efficient way to write unit tests. We will learn how to create test cases, execute them, and interpret the results to ensure our code is performing as expected. Making use of mocks is another essential concept that will be covered in this chapter. Mocks help us simulate the behavior of certain components or dependencies within our code, allowing us to focus solely on the unit being tested.

Understanding how to effectively utilize mocks is crucial for writing comprehensive and efficient unit tests.

Finally, we will discuss mastering debugging. Even the most skilled

developers encounter bugs from time to time. Being able to identify, isolate, and resolve these issues efficiently is what sets great developers apart. We will explore various debugging techniques and tools that will empower you to become a more proficient problem solver. By the end of this chapter, you will gain the knowledge and skills necessary to write robust unit tests, use mocks effectively, and confidently tackle the debugging process.

Structure

This chapter covers the following topics:

- Unit testing with xUnit
- Making usage of mocks
- Mastering debugging
- Applying xUnit and mocks

Objectives

In this chapter, we will explore a structured journey through software testing and debugging. Explore unit testing's role, xUnit framework, and crafting effective tests. Dive into assertions, mocks, and advanced techniques for scalable test code. Master debugging fundamentals, tools, and best practices. Apply knowledge to real-world scenarios. Gain proficiency in xUnit for comprehensive tests. Become skilled in debugging, delivering higher-quality software. This comprehensive chapter covers everything from the basics to advanced strategies, ensuring you are equipped to tackle any testing or debugging challenge in your software development journey.

Unit testing with xUnit

- At the time of writing this book, xUnit is the most used test framework for .NET applications, it has a simple and efficient approach for developer to write unit tests. Unit tests created with xUnit are supposed to validate small pieces of isolated code, as class methods for example, and no matter how many times you execute the unit tests you must have the same result. We can understand better about unit testing with xUnit and its main functionalities below: **Test method structure:** xUnit tests are defined as public methods within test classes.

These methods are identified as test cases based on naming conventions, where method names must start with **Fact** or **Theory**. `Fact_TestMethod1` or `Theory_TestMethod1` are examples of test method naming.

- **Assertions:** xUnit has a wide range of built-in assertion methods, making it easier to validate your expected result against the real result from your test methods.
- **Theory and data-driven testing:** xUnit supports data-driven testing using the Theory attribute. This allows you to run the same test method with different input data, enabling you to validate different scenarios.
- **Test fixtures:** You may make usage of xUnit's test fixtures, which are block of codes that can run before and, or, after your test cases. This allows you to configure required dependencies before test cases run and to clean used resources after finishing tests assertions.
- **Test collections:** xUnit allows you to group related test classes into collections. This can be beneficial when dealing with tests that share a common setup or need to run in a specific order.
- **Test output and logging:** xUnit provides features to capture and report test output and logging. This is useful for understanding what happened during test execution.
- **Parallel test execution:** with xUnit you can run different tests in parallel, increasing the speed of the test execution and by having faster output during software changes.
- **Extensibility:** xUnit is designed to be extensible, allowing you to create custom test runners, reporters, and other extensions to tailor the testing framework to your specific needs.
- **Test discovery and execution:** xUnit uses reflection to discover and execute test methods automatically. Test discovery locates all test methods in the test assembly, and test execution runs them, reporting the results.
- **Integration with continuous integration (CI):** xUnit integrates well with CI tools, making it easy to incorporate automated testing into your development pipeline.

Unit testing with xUnit promotes a **test-driven development (TDD)** approach, where developers write tests before implementing the corresponding functionality. By following this practice,

developers can ensure that their code meets the desired requirements and is less prone to bugs. xUnit's clean and straightforward syntax, combined with its powerful features, makes it an excellent choice for developers looking to create reliable and maintainable unit tests for their .NET applications.

Making usage of mocks

In software testing, mocks are objects that simulate the behavior of real dependencies, such as external services, databases, or complex components, during unit testing. Mocks allow developers to isolate the unit being tested and focus solely on its behavior without having to involve the actual dependencies. Here is an overview of how mocks are used in testing:

- **Isolating dependencies:** When testing a unit, it is essential to isolate it from its dependencies to ensure that the test focuses on the unit's logic in isolation. Mocks act as stand-ins for these dependencies, providing controlled responses that mimic the behavior of the real dependencies without actually using them.
- **Removing external dependencies:** Using real dependencies, especially external services or databases, in unit tests can make the tests slow, unreliable, and difficult to control. By replacing these dependencies with mocks, tests become faster, deterministic, and independent of external factors.
- **Controlled behavior:** Mocks are set up to provide predetermined responses to specific inputs, allowing developers to control the behavior of the mocked dependencies. This ensures that the unit being tested is exposed to various scenarios, allowing comprehensive test coverage.
- **Eliminating side effects:** Some dependencies may have side effects that are undesirable during testing. For example, sending actual emails or writing to a real database. Mocks prevent these side effects from occurring, making the tests more predictable and repeatable.
- **Test-driven development:** Mocks are commonly used in TDD practices, where developers write tests before implementing the actual functionality. By using mocks, developers can create tests for components that have not been developed yet,

enabling them to work incrementally on the system.

- **Integration testing vs. unit testing:** In integration testing, real dependencies are often used to verify the interactions between different components. However, in unit testing, mocks are preferred to focus on the isolated behavior of individual units.
- **Mocking frameworks:** To facilitate the creation and management of mocks, developers often use mocking frameworks like Moq (for C#), Mockito (for Java), or Sinon (for JavaScript). These frameworks provide APIs to define mock behavior and verify how the mocks were used during testing.
- **Verifying calls:** Besides setting up behavior, mocks can also be used to verify that certain methods were called with specific arguments. This allows developers to ensure that the unit under test interacts correctly with its dependencies.
- **Test code simplicity:** By providing controlled responses and avoiding complex setups, mocks simplify the test code and make it easier to read, write, and maintain.
- **Limitations of mocks:** While mocks are powerful tools for unit testing, they do have limitations. Overusing mocks can lead to brittle tests that tightly couple to implementation details, reducing the effectiveness of tests as the code evolves.

When used wisely, mocks enhance the effectiveness of unit testing by isolating units, making tests faster, and providing controlled behavior for dependencies. By employing mocks alongside real unit tests, developers can achieve a well-rounded testing strategy that improves software quality and stability.

Mastering debugging

Debugging is the process of identifying, analyzing, and resolving issues or bugs within software code. It is a crucial skill for software developers to ensure the reliability, correctness, and performance of their applications. Debugging involves systematically investigating the source of unexpected behavior or errors and finding solutions to correct them. Here is an overview of the debugging process:

- **Identifying the issue:** The first step in debugging is recognizing that there is a problem. This may be through user-

reported issues, error messages, unexpected behavior, or failing test cases. Understanding the symptoms and gathering relevant information is vital for effective debugging.

- **Reproducing the problem:** Once the issue is identified, developers need to reproduce the problem consistently. This involves identifying the steps or conditions that trigger the bug and replicating them in a controlled environment. Reproduction helps ensure that developers can verify the issue and test potential fixes.
- **Inspecting the code:** With the problem reproducible, developers examine the code related to the issue. This involves carefully inspecting the affected code and looking for logical errors, incorrect assumptions, or unintended consequences.
- **Using debugging tools:** Debugging is often facilitated by using specialized debugging tools provided by **integrated development environments (IDEs)** or language-specific debuggers. These tools allow developers to set breakpoints, inspect variable values, step through code execution, and observe the program's state during runtime.
- **Setting breakpoints:** Breakpoints are markers placed in the code to pause its execution at specific points. When the program reaches a breakpoint, developers can examine the current state of variables and the call stack, helping them understand the flow of the program.
- **Stepping through code:** Debuggers allow developers to step through the code one line at a time, either forward (step into) or over (step over) function calls. This helps to observe how the code behaves and identify the location of the issue.
- **Inspecting variables:** During debugging, developers can inspect the values of variables at runtime to identify unexpected or incorrect data, which may be causing the problem.
- **Fixing the issue:** Once the root cause of the problem is identified, developers work on implementing a solution. This may involve correcting logical errors, adjusting algorithmic approaches, or fixing implementation mistakes.
- **Regression testing:** After implementing the fix, developers run regression tests to ensure that the changes have not introduced new issues and that the original problem has been resolved.

- **Continuous improvement:** Debugging is an iterative process, and developers continuously improve their debugging skills. They learn from previous debugging experiences, adopt best practices, and seek to write more robust code to minimize future issues.

Debugging is an essential part of the software development lifecycle and requires a combination of technical skills, analytical thinking, and attention to detail. Effective debugging leads to improved software quality, reduced maintenance efforts, and higher customer satisfaction. It is a skill that developers continually refine and leverage to create reliable and resilient software systems.

Applying xUnit and mocks

In this practical example, we are implementing xUnit and mocks in the project created in [Chapter 1, Introduction to Visual Studio 2022](#):

1. Let us start with adding the xUnit project to the solution as depicted in the following figure:

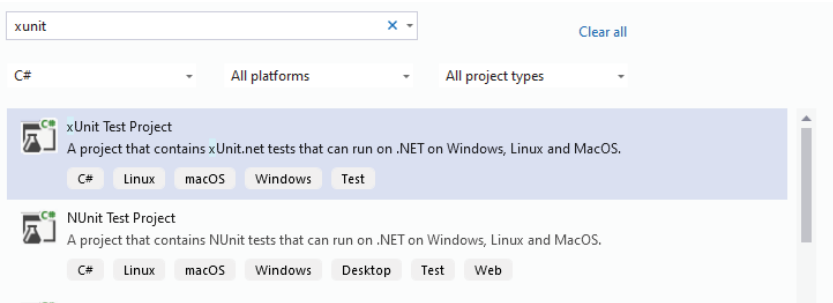


Figure 14.1: Visual Studio xUnit template project

1. Now, we add the following Nuget Package to our project:

<https://www.nuget.org/packages/Moq>

Also, an interface was added to the original project as follows:

```
1. public interface ICalculate
2. {
3.     int Sum();
4. }
```

Our **Calculate** class was updated to inherit from the

interface, as follows:

```
1. public class Calculate : ICalculate
2. {
3.     public Calculate(int numberA, int
numberB)
4.     {
5.         this.NumberA = numberA;
6.         this.NumberB = numberB;
7.
8.     }
9.
10.    private int NumberA { get; set; }
11.    private int NumberB { get; set; }
12.
13.    public int Sum() { return NumberA +
NumberB; }
14. }
```

Our Unit Tests using xUnit are the ones as follows: the first unit test is a normal test, whereas we create an instance of the **calculate** class. The second one makes use of mock, mocking the interface and setting up the expected output. Also, we make use of the **InlineData** attribute. The unit test class code is as follows:

```
1. public class UnitTest1
2. {
3.     [Fact]
4.     public void Test()
5.     {
6.         //arrange
7.         Calculate calculate = null;
8.
9.         //act
10.        calculate = new Calculate(2, 5);
11.        var result = calculate.Sum();
12.
```

```

13.         //assert
14.         Assert.Equal(7, result);
15.     }
16.
17.     [Theory]
18.     [InlineData(1, 8)]
19.     [InlineData(5, 2)]
20.     public void Test2(int a, int b)
21.     {
22.         //arrange
23.         Mock<ICalculate> calculate = new
Mock<ICalculate>();
24.         calculate.Setup(x =>
x.Sum()).Returns(a + b);
25.
26.         //act
27.         var result =
calculate.Object.Sum();
28.
29.         //assert
30.         Assert.Equal(a + b, result);
31.     }
32. }

```

1. If we run our unit tests, we have successfully output the following figure:

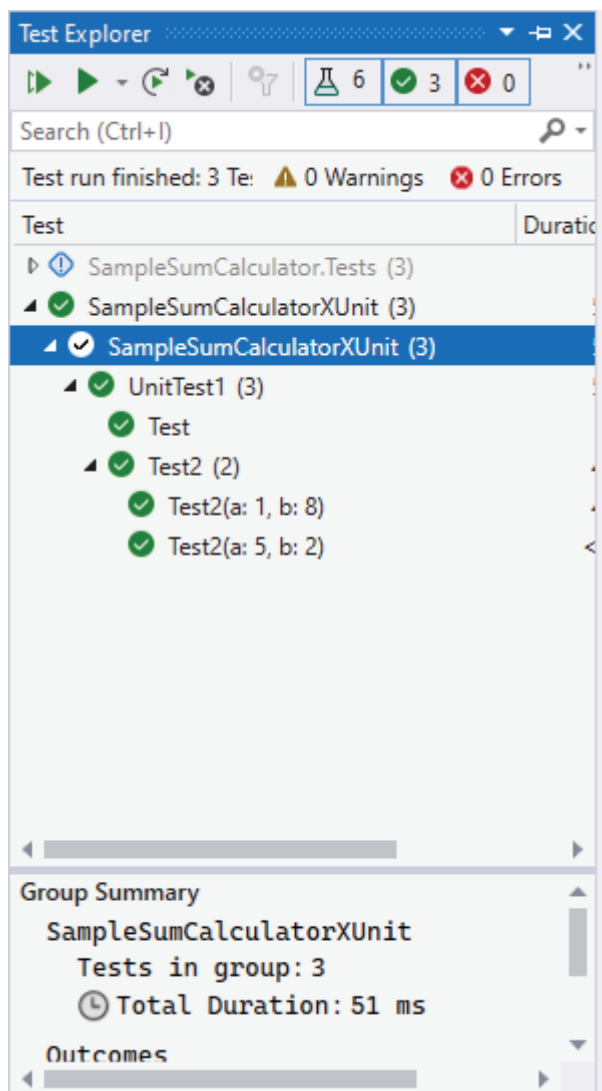


Figure 14.2: Visual Studio test explorer window

You may also debug your unit tests methods. On Visual Studio, the shortcut to do so is *Ctrl R + Ctrl T*. You may see the unit test being debugged as the following figure shows:

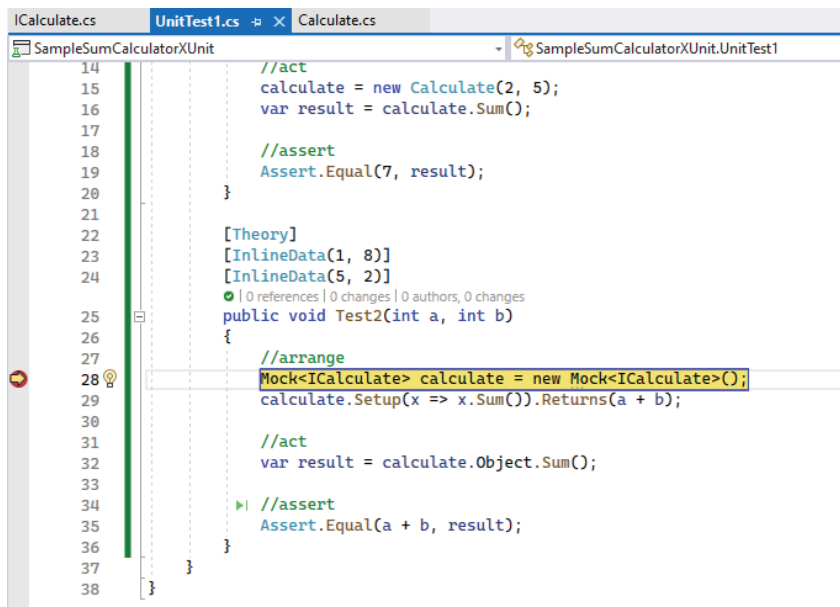


Figure 14.3: Unit test method being debugged

Conclusion

We hope you found this journey through the world of testing and debugging both insightful and practical. Unit testing with xUnit has equipped you with the ability to write automated tests that verify the correctness of individual units in your code. You now understand the importance of testing in the software development lifecycle and have the tools to build a suite of tests that will provide you with confidence in your codebase.

The knowledge of using mocks has expanded your testing horizons, enabling you to isolate components and dependencies when writing unit tests. This technique empowers you to create more focused and efficient tests that can adapt to different scenarios.

Mastering debugging is an invaluable skill that you have developed. You can now skillfully identify, diagnose, and resolve issues within your code, ensuring that your software functions as intended and delivers the expected results to users. Remember that testing and debugging are continuous processes in the life of a developer. Keep honing your skills, exploring new testing methodologies, and

staying up-to-date with the latest tools and practices in the field.

As you move forward in your software development journey, always remember that reliable code is the backbone of exceptional software products. Embrace testing and debugging as integral parts of your development workflow, and they will serve as the pillars of your success.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



Index

A

Advanced Message Queuing Protocol (AMQP) [255](#)

ASP.NET [175](#)

authentication, SignalR

 bearer token authentication [219](#)

 cookie authentication [219](#)

 identity server JWT authentication [220](#)

 Windows authentication [221](#)

Azure DevOps [291](#)

 features [291](#), [292](#)

Azure Function [79](#), [80](#)

 benefits [81](#), [82](#)

 case study [84](#)

 creating [84-86](#)

 output, testing [94-96](#)

Azure Function bindings [83](#)

 input bindings [83](#)

 output bindings [83](#)

 selecting [88-93](#)

 trigger bindings [83](#)

Azure Function triggers [82](#)

 blob trigger [82](#)

 Cosmos DB trigger [82](#)

 event grid trigger [82](#)

 event hub trigger [82](#)

 HTTP trigger [82](#)

 queue trigger [82](#)

 selecting [86](#), [87](#)

 service bus trigger [82](#)

 timer trigger [82](#)

Azure Key Vault [161](#)

 access policies, managing [168](#)

 authentication [163](#)

 Azure AD authentication [164](#)

 case study [166](#)

 certificate authentication [164](#)

 creating [166](#), [167](#)

 features [163](#)

- key accessing [169-173](#)
- managed identity authentication [164](#)
- overview [162](#)
- policies [165](#), [166](#)
- SAS authentication [164](#)
- service principal authentication [164](#)
- Azure Service Bus [130](#)
 - async operations [130](#)
 - case study [137](#), [138](#)
 - components [131](#)
 - creating [138-143](#)
 - features [131](#)
 - message batches, consuming [148](#), [149](#)
 - message processor [150-152](#)
 - messages, consuming [146-148](#)
 - publishing [143-146](#)
 - session processor [153-156](#)
 - sessions, consuming [152](#), [153](#)
 - topics and subscriptions, consuming [156](#), [157](#)
 - versus, Azure Queue Storage Queues [136](#), [137](#)
- Azure Service Bus Queues [132](#)
 - characteristics [132](#), [133](#)
 - features [132](#)
 - Session Queues [133](#)
- Azure Service Bus Subscriptions [135](#)
 - characteristics [135](#)
 - features [135](#)
- Azure Service Bus Topics [133](#)
 - characteristics [133](#), [134](#)
 - features [133](#), [134](#)
- Azure SQL [99](#), [101](#)
 - database, connecting to [105-109](#)
 - examples [101](#)
 - features [101](#)
 - server, creating [103-105](#)
 - server scaling [102](#), [103](#)
 - usage example [103](#)

B

- Backend for Frontend (BFF) [261](#)
 - benefits [261](#), [262](#)
 - case study [265-278](#)
- Blazor [175](#), [176](#)
 - authorization and authentication [192-201](#)
 - best practices [184](#), [185](#)
 - practical case study [185](#), [186](#)
 - versus, Blazor Hybrid [309](#), [310](#)
 - versus, Razor [184](#)

- Blazor Hybrid [307](#)
- Blazor Server App project
 - creating [186-189](#)
- Blazor WebAssembly apps [180](#)
- Blob Storage [100](#), [120](#)
 - Azure resource, creating [123-125](#)
 - database, connecting [125-127](#)
 - examples [121](#)
 - features [121](#)
 - scaling [122](#), [123](#)
 - usage example [123](#)

C

- C# 11 [19](#)
- C# 11 updates [21](#)
 - auto-default structs [34-36](#)
 - extended nameof scope [37](#)
 - file-local types [30-32](#)
 - generic attributes [24](#)
 - generic math support [22](#)
 - IntPtr [38](#)
 - list patterns [27-30](#)
 - method group conversion [40,-42](#)
 - newlines, in string interpolation expressions [25-27](#)
 - pattern match Span<char>, on constant string [36](#)
 - raw string literals [21](#)
 - ref fields [38](#), [39](#)
 - required members [32-34](#)
 - scoped ref [38](#), [39](#)
 - UIntPtr [38](#)
 - UTF-8 string literals [25](#)
 - warning wave [42](#), [43](#)
- C# 12 [19](#)
- C# 12 updates [43](#)
 - alias any type [48](#)
 - collection expression [44](#)
 - default lambda parameters [45](#), [46](#)
 - inline arrays [45](#)
 - primary constructors [43](#)
 - ref readonly parameters [47](#), [48](#)
- CI/CD pipeline
 - case study [294](#), [295](#)
 - with Docker, in Azure DevOps [282](#), [283](#)
- client configuration options, SignalR [215](#)
 - additional options [216-218](#)
 - allowed transports configuration [215](#)
 - bearer authentication configuration [216](#)
 - logging configuration [215](#)

- Code First [55](#)
 - benefits [55](#)
 - implementation [55-57](#)
- Command Query Responsibility Segregation (CQRS) [261](#), [262](#)
 - Command Model [262](#)
 - Query Model [263](#)
- Commit Graph [3](#)
- Common Language Runtime (CLR) [59](#)
- continuous deployment [293](#)
 - applying [304](#), [305](#)
 - benefits [293](#), [294](#)
- continuous integration [292](#)
 - applying [298-303](#)
 - benefits [292](#), [293](#)
- Cosmos DB [99](#), [109](#)
 - account, creating [113](#), [114](#)
 - change feed notifications service [112](#), [113](#)
 - Cosmos Containers [110](#)
 - database, connecting to [115-120](#)
 - examples [110](#)
 - features [109](#), [110](#)
 - scaling [111](#)
 - stored procedures [112](#)
 - triggers [111](#), [112](#)
 - usage examples, for NoSQL [113](#)
 - User-Defined Functions (UDFs) [112](#)
- CRUD operations [70](#)
- cross-platform application development [310](#)
 - Blazor Hybrid UI, using from Desktop client [312](#), [313](#)
 - Blazor Hybrid UI, using from mobile client [313-316](#)
 - .NET MAUI project, creating [310](#), [311](#)

D

- Data Annotations [59](#)
 - applying [59-61](#)
 - common annotations [59](#)
- Database First approach [52](#), [53](#)
 - advantages [53](#)
 - implementation [53](#), [54](#)
- data binding [182](#), [183](#)
 - chained data binding [183](#), [184](#)
 - example [201-204](#)
 - two-way data binding [183](#)
- Data Management [70](#)
 - normal usage [70](#)
 - repository pattern [74-76](#)
 - unit of work pattern [72](#)
- Data Modeling [61](#)

- many-to-many relationship [68](#), [69](#)
- one-to-many relationship [65](#)
- one-to-one relationship [62](#)
- DbContext class [105](#)
- debugging [335](#), [338](#)
 - overview [339](#)
- Denial-of-Service (DoS) attack [213](#)
- Discard Pattern [28](#)
- Docker [283](#), [284](#)
 - advantages, for apps [287](#)
- Docker commands [290](#)
 - docker build [290](#)
 - docker image [291](#)
 - docker ps [290](#)
 - docker rm [291](#)
 - docker rmi [291](#)
 - docker run [290](#)
 - docker stop [290](#)
- Docker container [284](#)
 - functionalities [284](#)
- Docker container image [284](#)
 - benefits [285](#)
- Dockerfile [287-289](#)
 - creating [295](#), [296](#)
 - for multi-stage builds [289](#), [290](#)
- Docker images
 - benefits [285](#), [286](#)
 - creating [296](#), [297](#)
 - running [297](#), [298](#)
- Document Object Model (DOM) elements [183](#)
- Domain Driven Design (DDD) [261](#), [264](#)
 - principles [264](#), [265](#)

E

- Entity Framework Core (EF Core) [52](#)
 - mastering [52](#)

F

- First-In-First-Out (FIFO) approach [129](#)
- Fluent Design System [319](#)
 - applications [320](#)
 - key principles [319](#), [320](#)

H

- horizontal scalability [256](#)
 - benefits [257](#)

- considerations [258](#)
- hot reload
 - working [190](#), [191](#)

I

- integrated development environments (IDEs) [339](#)
- IntelliCode [4](#)
- Internet of Things (IoT) messaging [132](#)

L

- Language Integrated Query (LINQ) [26](#), [51](#), [70](#)
- LINQ to Entities [58](#), [59](#)
- Live Unit Testing [7](#)
 - customizing, according to needs [10](#)
 - supported testing frameworks [9](#)
 - test methods, excluding and including [9](#), [10](#)
 - test projects, excluding and including [9](#), [10](#)
 - using [7](#), [8](#)

M

- many-to-many relationship [68](#), [69](#)
- MessagePack [211](#)
- Message Queuing Telemetry Transport (MQTT) [255](#)
- message sessions [152](#)
- microservices
 - architectural patterns [261](#)
 - asynchronous communication [251](#)
 - implementing, with WebAPIs [250](#), [251](#)
- microservices scalability [256](#)
 - horizontal scalability [256-258](#)
 - orchestrators [259](#), [260](#)
 - vertical scalability [256-259](#)
- Microsoft Design Language 2 [319](#)
- mocks [337](#)
 - applying [340-342](#)
 - using, in testing [337](#), [338](#)

N

- .NET hot reload [177](#)
 - configuring [179](#), [180](#)
 - supported frameworks and application types [177](#), [178](#)
 - unsupported scenarios [178](#), [179](#)
- .NET MAUI [307](#)
 - components [309](#)
 - overview [308](#)

O

- object-relational mapping (ORM) tool [51](#)
- one-to-many relationship [65](#)
 - optional one-to-many [66](#), [67](#)
 - required one-to-many [65](#), [66](#)
- one-to-one relationship [62](#)
 - optional one-to-one [63](#), [64](#)
 - required one-to-one [62](#), [63](#)
- orchestrators [259](#), [260](#)

P

- Program Synthesis using Examples (PROSE) [4](#)

R

- RabbitMQ [251](#), [252](#)
 - benefits [254](#), [255](#)
 - features [252-254](#)
- raw string literals [21](#)
- Razor [184](#)
- repository pattern [74-76](#)

S

- SampleThiagoDb [53](#)
- scaling out [256](#)
- security, Blazor [180](#), [181](#)
 - authorization [182](#)
 - Authorize attribute [182](#)
 - AuthorizeView component [182](#)
 - Blazor server authentication [181](#)
 - Blazor WebAssembly authentication [181](#)
- Session Queues [133](#)
- SignalR [208](#)
 - advanced HTTP configuration [213-215](#)
 - authentication [219](#), [232-242](#)
 - authorization [219](#), [232-42](#)
 - authorization handlers [222](#), [223](#)
 - case study [224-232](#)
 - claims [221](#)
 - client configuration options [215](#)
 - configuration [210](#), [211](#)
 - custom authorization policy [243-247](#)
 - examples [209](#)
 - hubs [209](#), [210](#)
 - hubs and hubs methods authorization [222](#)
 - JSON encoding [211](#)
 - MessagePack encoding [211](#)

- message transports [209](#)
- methods [210](#)
- real-time communication [208](#)
- server configuration options [211-213](#)
- Snapshot Debugger [11](#)
- snapshot debugging [11](#)
 - required permissions and limitations [15](#)
 - supported frameworks and environments [14, 15](#)
 - using [11-14](#)
- SQL Server Management Studio (SSMS) [104](#)
- streaming hub [223](#)
 - client-to-server streaming hub [224](#)
 - server-to-client streaming hub [223](#)

T

- Team Foundation Version Control (TFVC) [291](#)
- test-driven development (TDD) [337](#)
- Time-to-Live (TTL) value [132](#)
- Time Travel Debugging (TTD) [15](#)
 - limitation [17](#)
 - snapshot, recording [16, 17](#)
 - snapshot, viewing [17](#)
 - using [16](#)

U

- unit of work patter [72](#)
- unit testing [335](#)
 - with xUnit [336, 337](#)
- Universal Windows Platform (UWP) apps [320](#)

V

- Var Pattern [30](#)
- vertical scalability [256-258](#)
 - benefits [259](#)
 - limitations [259](#)
- Visual Studio 2022 [2](#)
 - 64 -bit support [3, 4](#)
 - Commit Graph [3](#)
 - highlights [6, 7](#)
 - hot reload, with new capabilities [5](#)
 - interactive staging support [6](#)
 - interface improvements and customization [6](#)
 - multi-repository support [5](#)
 - performance improvements [2, 3](#)
 - Razor editor improvements [4, 5](#)
 - Smarter IntelliCode [4](#)

W

Web APIs [249](#)

microservices, implementing with [250](#)

Web apps

with Blazor and .NET [176](#)

WinUI [318](#)

advantages [318](#), [319](#)

case study [320](#)

WinUI project

cache, implementing [323](#), [324](#)

creating [321](#)

data transfer between pages [324-333](#)

user interface, designing [322](#), [323](#)

X

xUnit [336](#)

applying [340-342](#)

unit testing with [336](#)